

renren-fast开发文档_完整版

1. 介绍

1.1. 项目描述

- **renren-fast**是一个轻量级的 `Spring Boot` 快速开发平台，能快速开发项目并交付【接私活利器】
 - 完善的 `xss` 防范及脚本过滤，彻底杜绝 `xss` 攻击
 - 实现前后端分离，通过 `token` 进行数据交互
 - 推荐使用阿里云服务器部署项目，免费领取阿里云优惠券，请点击【[免费领取](#)】
-

1.2. 获取帮助

- Git地址：<http://git.oschina.net/babaio/renren-fast>
 - 代码生成器：<http://git.oschina.net/babaio/renren-generator>
 - 其他项目：<http://www.renren.io/open/>
 - 如需关注项目最新动态，请Watch、Star项目，同时也是对项目最好的支持
-

1.3. 官方QQ群

- 高级群：324780204(大牛云集，跟大牛学习新技能)
 - 普通群：145799952(学习交流，互相解答各种疑问)
-

2. 入门

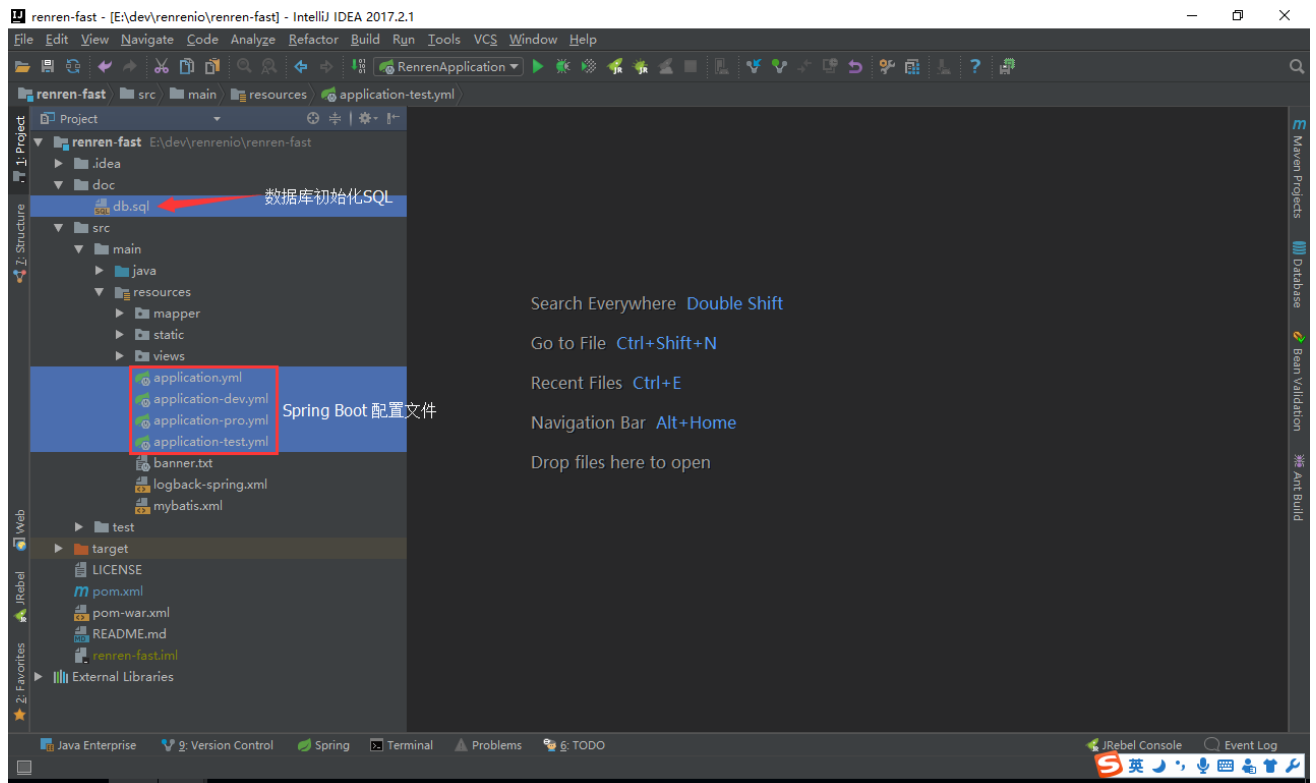
2.1. 快速开始

2.1.1. 本地部署

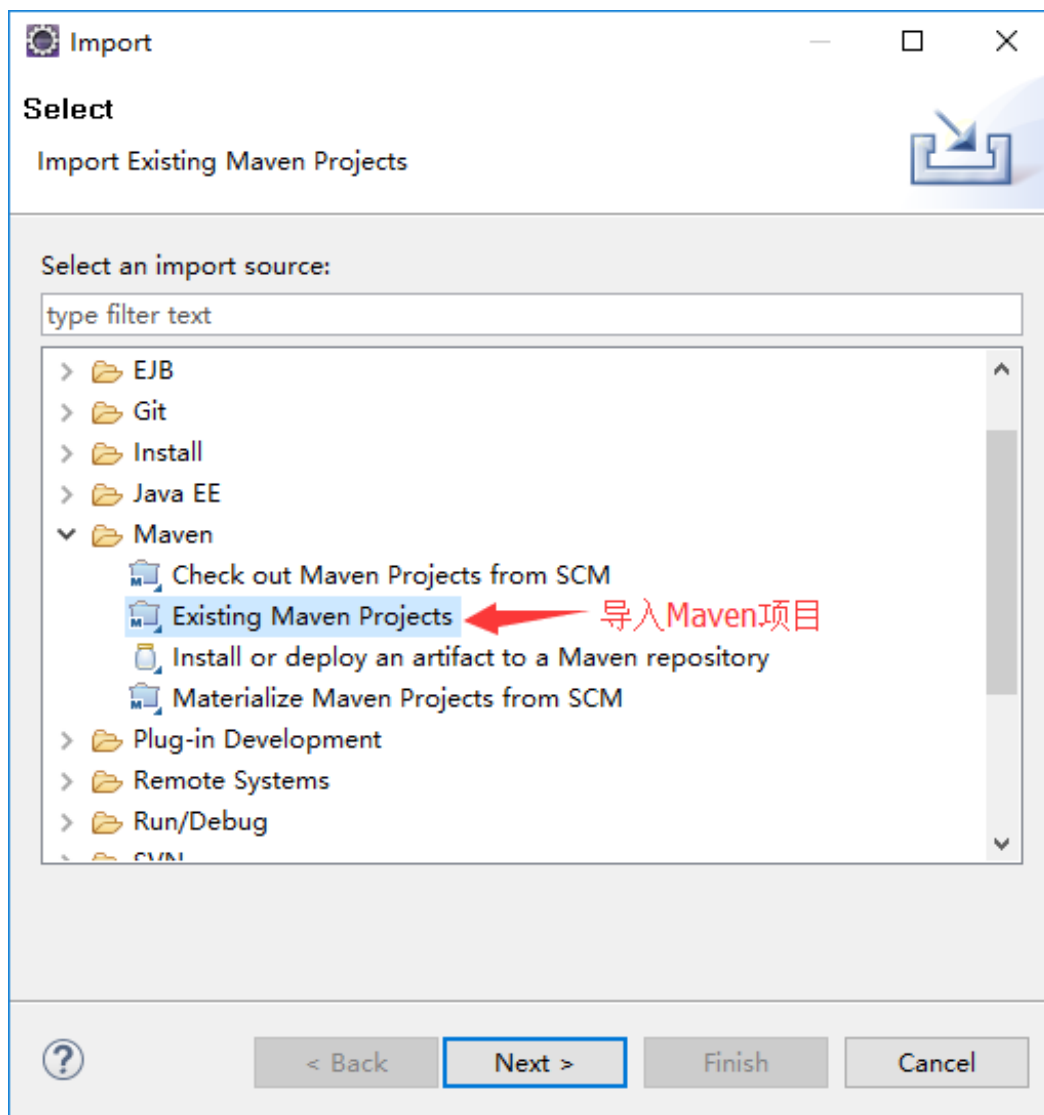
- 环境要求 JDK1.8、Tomcat8.0+、MySQL5.5+
- 通过 git , 下载renren-fast源码, 如下:

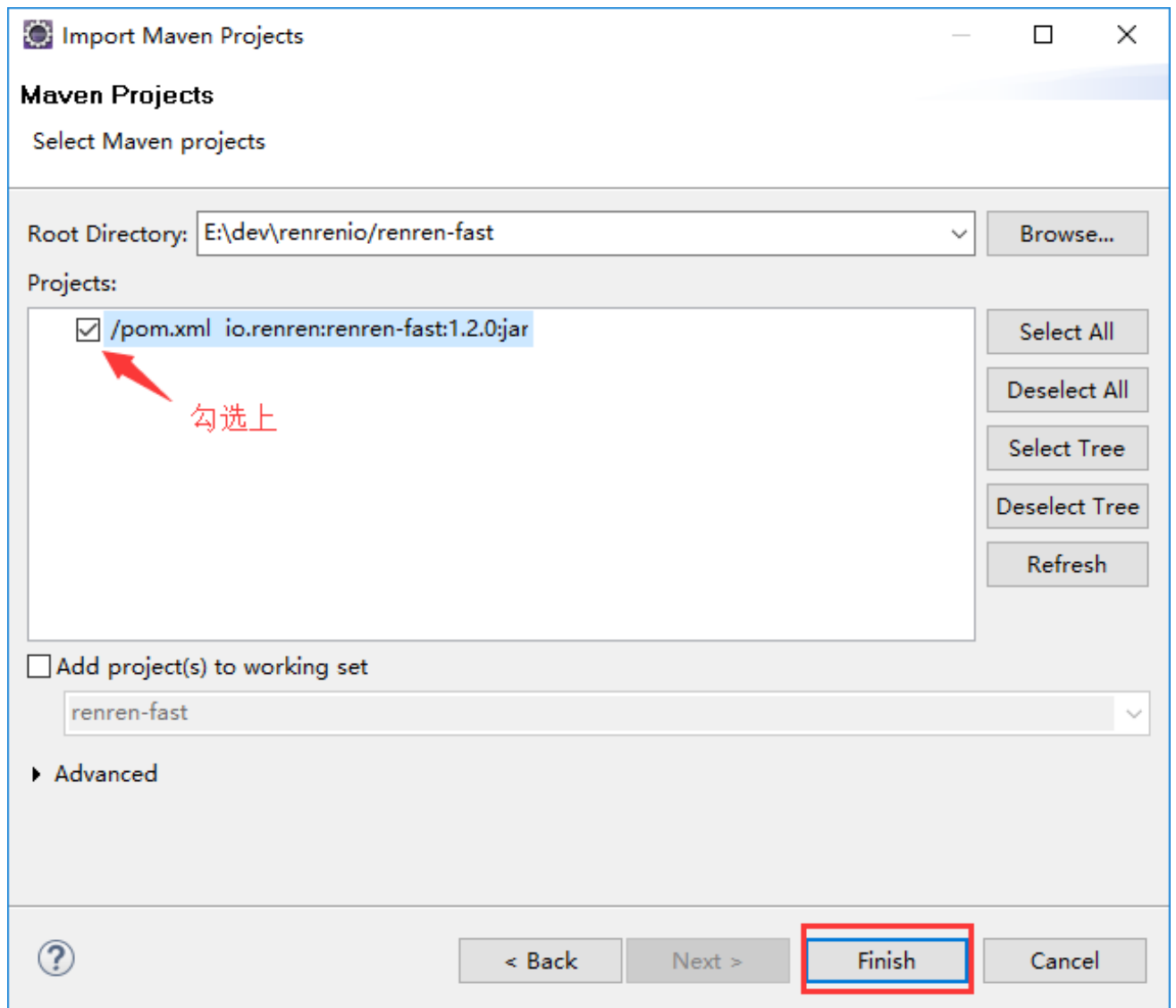
```
1. git clone https://git.oschina.net/babaio/renren-fast.git
```

- 用 idea 打开项目, File -> Open 如下图:



- 用 eclipse 打开项目, 如下图:





- 创建数据库 `renren_fast`，数据库编码为 `UTF-8`
- 执行 `doc/db.sql` 文件，初始化数据
- 修改 `application-dev.yml`，更新MySQL账号和密码
- 运行 `io.renren.RenrenApplication.java` 的 `main` 方法，则可启动项目
- 项目访问路径：<http://localhost:8080/renren-fast>
- 账号密码：admin/admin
- Swagger路径：<http://localhost:8080/renren-fast/swagger/index.html>

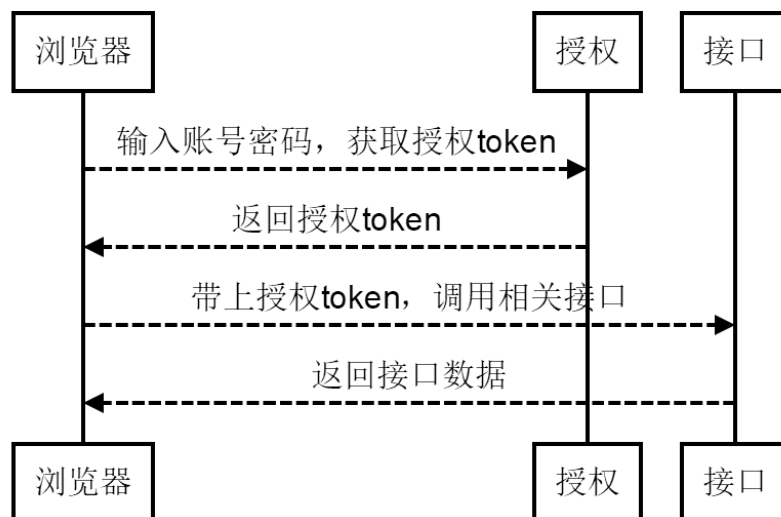
2.1.2. 配置文件

```
1.  # Tomcat
2.  server:
3.      tomcat:
4.          uri-encoding: UTF-8
5.          max-threads: 1000
6.          min-spare-threads: 30
7.          port: 8080  #tomcat端口号
8.          context-path: /renren-fast
9.
10. spring:
11.     # 环境 dev|test|pro
12.     profiles:
13.         active: dev
14.     # jackson时间格式化
15.     jackson:
16.         time-zone: GMT+8
17.         date-format: yyyy-MM-dd HH:mm:ss
18.     http:
19.         multipart:
20.             max-file-size: 100MB
21.             max-request-size: 100MB
22.             enabled: true
23.     resources: # 指定静态资源的路径
24.         static-locations: classpath:/static/,classpath:/views/
25.     redis:
26.         open: false  # 是否开启redis缓存 true开启 false关闭
27.         database: 0
28.         host: redis.open.renren.io
29.         port: 16379
30.         password:      # 密码（默认为空）
31.         timeout: 6000  # 连接超时时长（毫秒）
32.         pool:
33.             max-active: 1000  # 连接池最大连接数（使用负值表示没有限制）
34.             max-wait: -1      # 连接池最大阻塞等待时间（使用负值表示没有限制）
35.             max-idle: 10      # 连接池中的最大空闲连接
36.             min-idle: 5       # 连接池中的最小空闲连接
37.
38.     # Mybatis配置
39.     mybatis:
40.         mapperLocations: classpath:mapper/**/*.xml
41.         configLocation: classpath:mybatis.xml
42.
43.     renren:
44.         # APP模块，是通过jwt认证的，如果要使用APP模块，则需要修改【加密密钥】
45.         jwt:
```

```
46.         # 加密秘钥
47.         secret: f4e2e52034348f86b67cde581c0f9eb5[www.renren.io]
48.         # token有效时长, 7天, 单位秒
49.         expire: 604800
50.         header: token
51.
52.     # 数据库配置
53.     spring:
54.         datasource:
55.             type: com.alibaba.druid.pool.DruidDataSource
56.             driverClassName: com.mysql.jdbc.Driver
57.             druid:
58.                 first: #数据源1
59.                     url: jdbc:mysql://localhost:3306/renren_fast?allowMulti
60.                         Queries=true&useUnicode=true&characterEncoding=UTF-8
61.                     username: renren
62.                     password: 123456
63.                 second: #数据源2
64.                     url: jdbc:mysql://localhost:3306/bdshop?
65.                         allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
66.                     username: renren
67.                     password: 123456
68.                     initial-size: 10
69.                     max-active: 100
70.                     min-idle: 10
71.                     max-wait: 60000
```

2.2. 数据交互

- 一般情况下，web项目都是通过session进行认证，每次请求数据时，都会把jsessionId放在cookie中，以便与服务端保持会话
- 本项目是前后端分离的，通过token进行认证（登录时，生成唯一的token凭证），每次请求数据时，都会把token放在header中，服务端解析token，并确定用户身份及用户权限，数据通过json交互
- 数据交互流程：



3. 项目实战

3.1. 功能描述

我们来完成一个商品的列表、添加、修改、删除功能，熟悉如何快速开发自己的业务功能模块。

- 我们先建一个商品表tb_goods，表结构如下所示：

```
1. CREATE TABLE `tb_goods` (  
2.     `goods_id` bigint NOT NULL AUTO_INCREMENT,  
3.     `name` varchar(50) COMMENT '商品名',  
4.     `intro` varchar(500) COMMENT '介绍',  
5.     `price` decimal(10,2) COMMENT '价格',  
6.     `num` int COMMENT '数量',  
7.     PRIMARY KEY (`goods_id`)  
8. ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='商品管理';
```

- 接下来，我们利用代码生成器，帮我们生成基础代码，可以大大的节省重复工作量，代码生成器Git地址：<https://git.oschina.net/babaio/renren-generator>

3.2. 使用代码生成器

- 代码生成器是Spring Boot开发的项目，通过git clone把项目下载到本地后，直接运行main方法，就可以通过<http://localhost>打开，我们先来看看配置文件，如下所示：

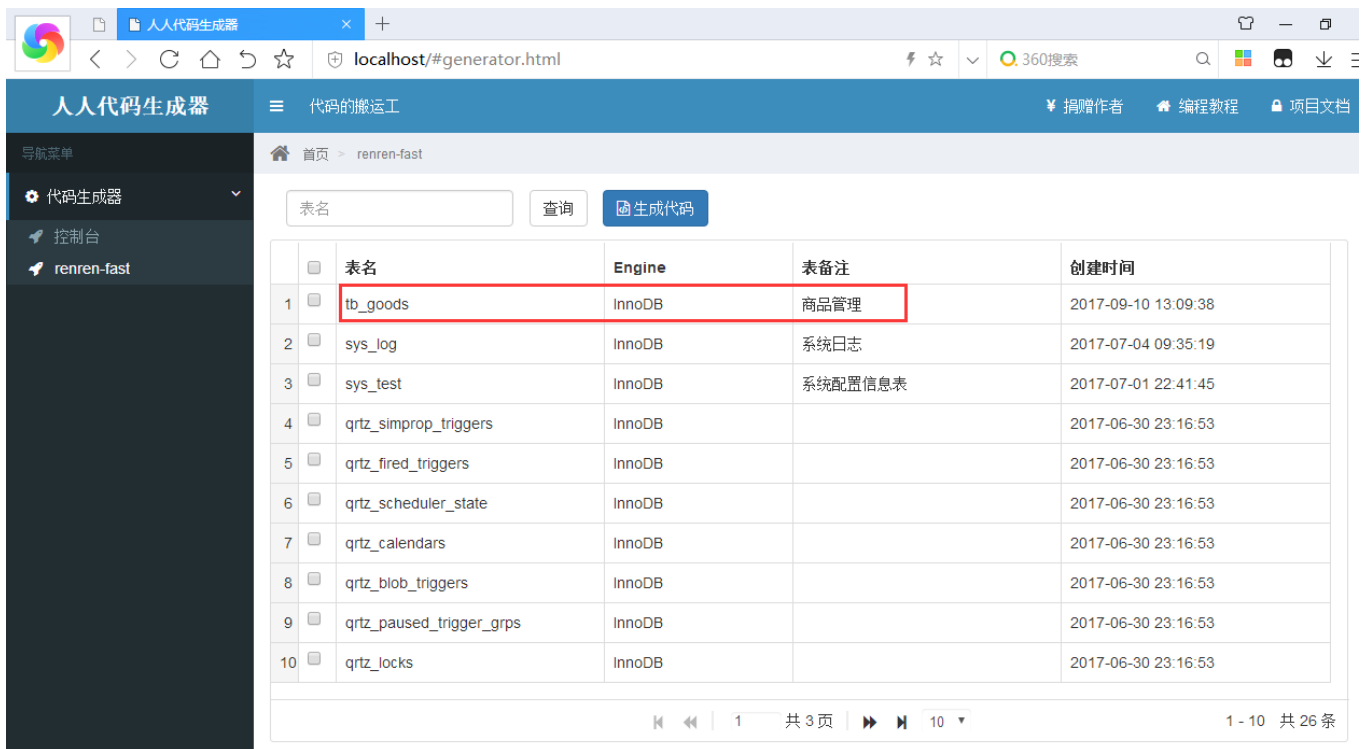
```
1.  #配置文件：generator.properties
2.
3.  #包名
4.  package=io.renren.modules.generator
5.  #作者
6.  author=chenshun
7.  #Email
8.  email=sunlightcs@gmail.com
9.  #表前缀 (类名不会包含表前缀)
10. tablePrefix=tb_
11.
12. #类型转换，配置信息
13. tinyint=Integer
14. smallint=Integer
15. mediumint=Integer
16. int=Integer
17. integer=Integer
18. bigint=Long
19. float=Float
20. double=Double
21. decimal=BigDecimal
22. bit=Boolean
23.
24. char=String
25. varchar=String
26. tinytext=String
27. text=String
28. mediumtext=String
29. longtext=String
30.
31. date=Date
32. datetime=Date
33. timestamp=Date
```

上面的配置文件，可以配置包名、作者信息、表前缀、类型转换。其中，类型转换是指，MySQL中的类型与JavaBean中的类型，是怎么一个对应关系。如果有缺少的类型，可自行在generator.properties文件中补充。

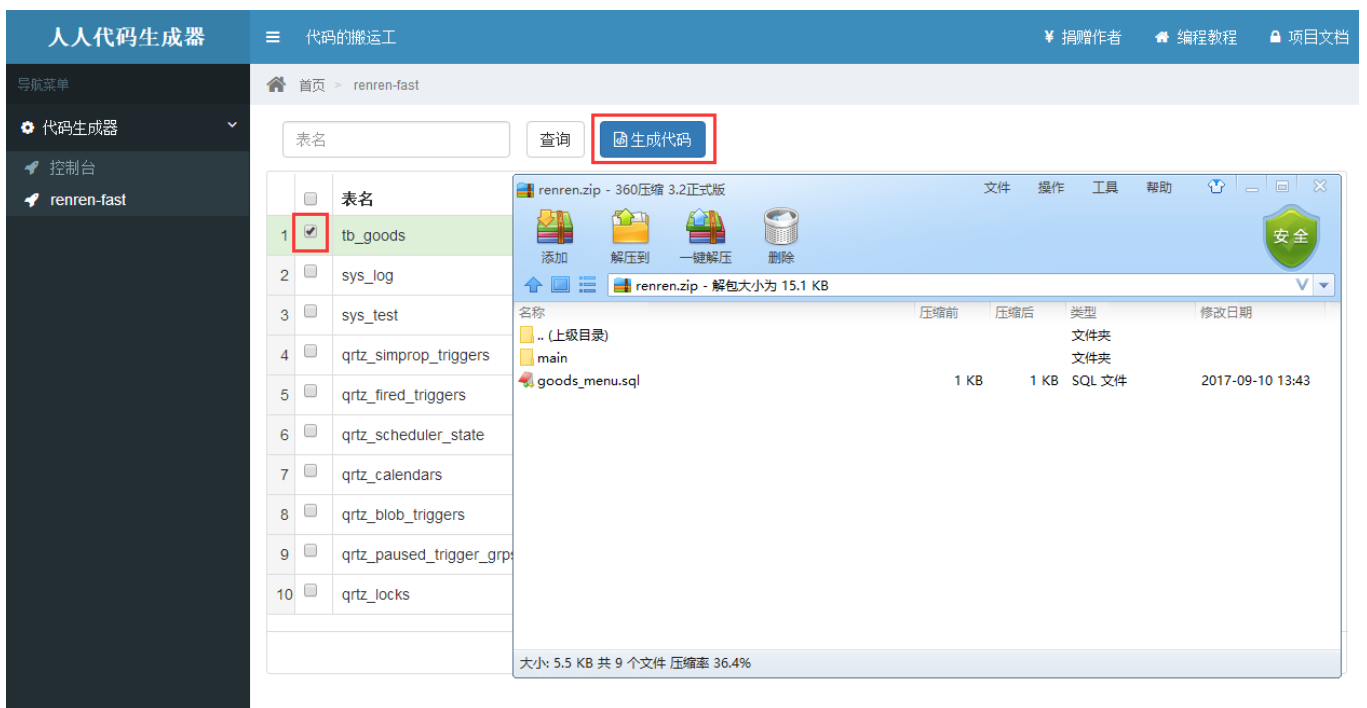
- 再看看application.yml配置文件，我们只要修改数据库名、账号、密码，就可以了。其中，数据库名是指待生成的表，所在的数据库。

```
1.  # Tomcat
2.  server:
3.      tomcat:
4.          max-threads: 10
5.          min-spare-threads: 10
6.          port: 80
7.
8.  # mysql
9.  spring:
10.     datasource:
11.         type: com.alibaba.druid.pool.DruidDataSource
12.         driverClassName: com.mysql.jdbc.Driver
13.         url: jdbc:mysql://localhost:3306/renren_fast?useUnicode=true&characterEncoding=UTF-8
14.         username: renren
15.         password: 123456
16.     jackson:
17.         time-zone: GMT+8
18.         date-format: yyyy-MM-dd HH:mm:ss
19.     resources:
20.         static-locations: classpath:/static/,classpath:/views/
21.
22.  # Mybatis配置
23.  mybatis:
24.     mapperLocations: classpath:mapper/**/*.xml
```

- 在数据库renren_fast中，执行建表语句，创建tb_goods表，再启动renren-generator项目(运行RenrenApplication.java的main方法即可)，如下所示：



- 我们只需勾选tb_goods，点击【生成代码】按钮，则可生成相应代码，如下所示：



3.3. 测试项目

- 我们先在renren_fast库中，执行goods_menu.sql语句，这个SQL是生成菜单的，SQL语句如下所示：

```

1.  -- 菜单SQL
2.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `i
   con`, `order_num`)
3.      VALUES ('1', '商品管理', 'modules/generator/goods.html', NULL, '1',
   'fa fa-file-code-o', '6');
4.
5.  -- 按钮父菜单ID
6.  set @parentId = @@identity;
7.
8.  -- 菜单对应按钮SQL
9.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `i
   con`, `order_num`)
10.      SELECT @parentId, '查看', null, 'goods:list,goods:info', '2', null,
   '6';
11.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `i
   con`, `order_num`)
12.      SELECT @parentId, '新增', null, 'goods:save', '2', null, '6';
13.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `i
   con`, `order_num`)
14.      SELECT @parentId, '修改', null, 'goods:update', '2', null, '6';
15.  INSERT INTO `sys_menu` (`parent_id`, `name`, `url`, `perms`, `type`, `i
   con`, `order_num`)
16.      SELECT @parentId, '删除', null, 'goods:delete', '2', null, '6';

```

- 然后把生成的main目录覆盖renren-fast的main目录，再重启renren-fast，效果如下所示：

人人快速开发平台

欢迎 admin

[捐赠作者](#)
[编程教程](#)
[修改密码](#)
[退出系统](#)

系统管理

管理员列表

角色管理

菜单管理

SQL监控

定时任务

参数管理

文件上传

商品管理

系统日志

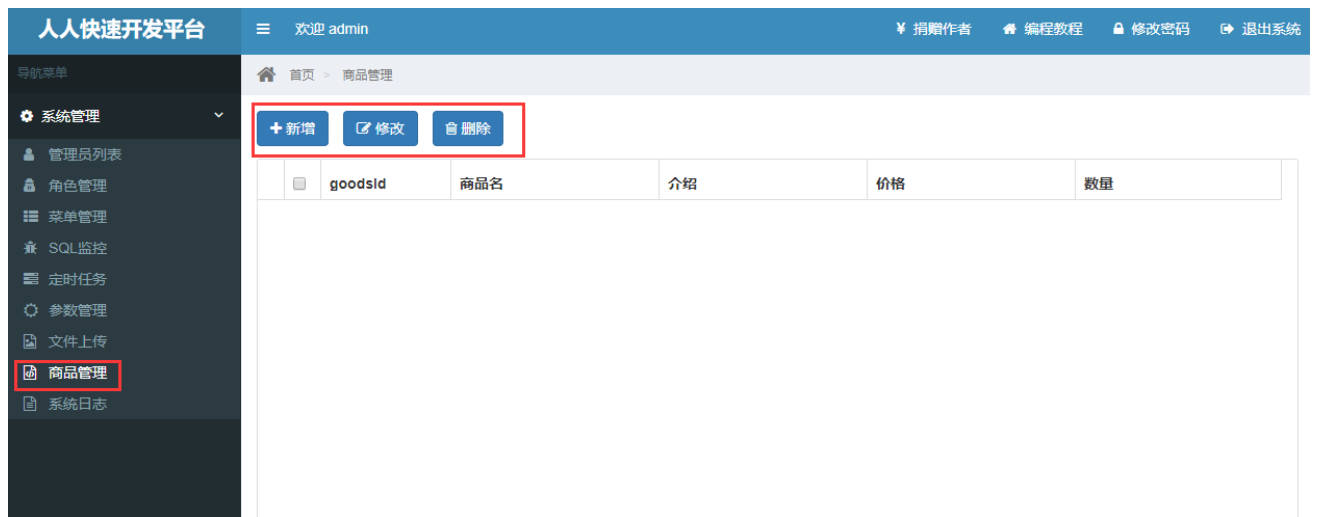
菜单管理

新增

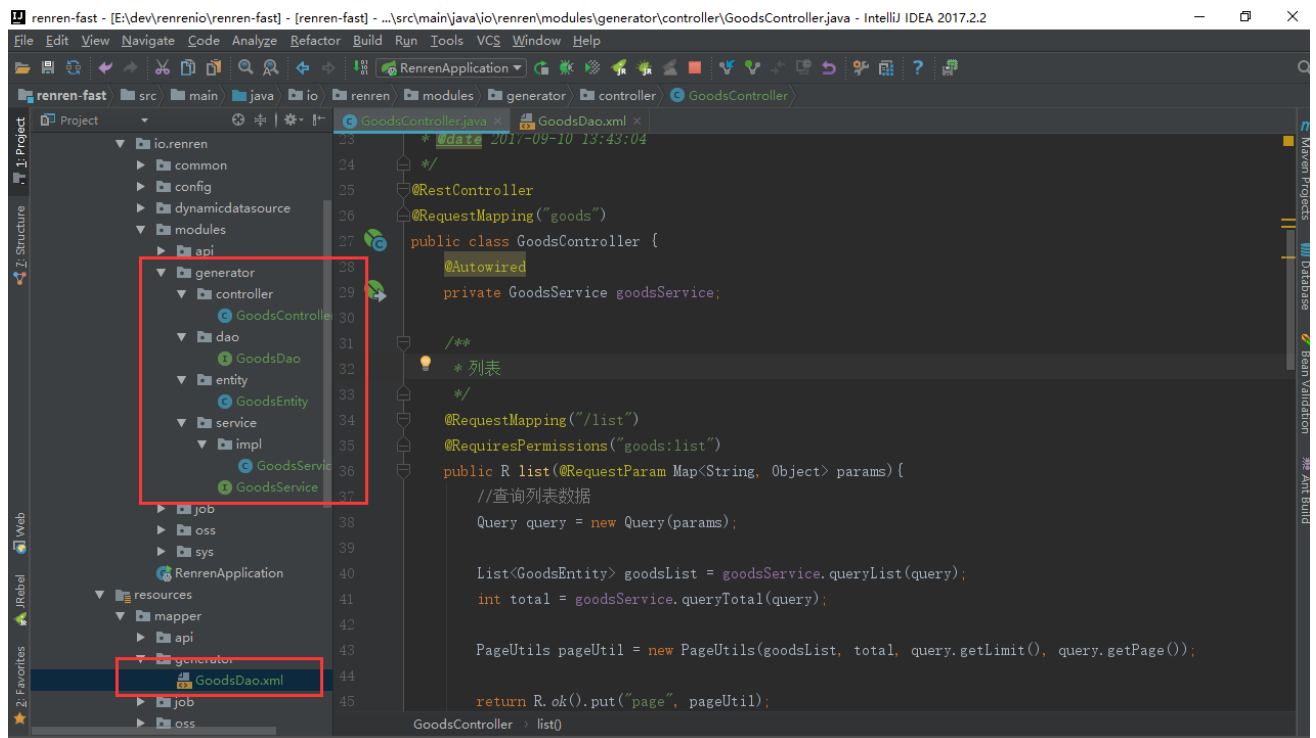
修改

删除

	菜单ID	菜单名称	上级菜单	图标	类型	排序号	菜单URL	授权标识
	4	菜单管理	系统管理		菜单	3	modules/sys/menu.h...	
	5	SQL监控	系统管理		菜单	4	druid/sql.html	
	6	定时任务	系统管理		菜单	5	modules/job/schedul...	
	27	参数管理	系统管理		菜单	6	modules/sys/config....	sys.config:list,sys:...
	30	文件上传	系统管理		菜单	6	modules/oss/oss.html	sys.oss:all
	31	商品管理	系统管理		菜单	6	modules/generator/g...	
	32	查看	商品管理		按钮	6		goods:list,goods:info
	33	新增	商品管理		按钮	6		goods:save
	34	修改	商品管理		按钮	6		goods:update
	35	删除	商品管理		按钮	6		goods:delete
	29	系统日志	系统管理		菜单	7	modules/sys/log.html	sys.log:list



- 我们就操作了这几步，就把查询、新增、修改、删除就完成了，是不是很快啊，下面就是我们才生成的代码，如下所示：



4. 源码分析

4.1. 多数据源

多数据源的应用场景，主要针对跨多个MySQL实例的情况；如果是同实例中的多个数据库，则没必要使用多数据源。

4.1.1. 实现多数据源

- 步骤1，在spring boot中，增加多数据源的配置，如下所示：

```
1.  spring:
2.     datasource:
3.         type: com.alibaba.druid.pool.DruidDataSource
4.         driverClassName: com.mysql.jdbc.Driver
5.         druid:
6.             first: #数据源1
7.                 url: jdbc:mysql://192.168.1.10:3306/renren_fast?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
8.                 username: renren
9.                 password: 123456
```

```

10.         second: #数据源2
11.             url: jdbc:mysql://192.168.1.11:3306/order?
allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
12.             username: root
13.             password: root

```

- 步骤2，扩展Spring的AbstractRoutingDataSource抽象类，AbstractRoutingDataSource中的抽象方法determineCurrentLookupKey是实现多数据源的核心，并对该方法进行Override，如下所示：

```

1.  public class DynamicDataSource extends AbstractRoutingDataSource {
2.      private static final ThreadLocal<String> contextHolder = new Thread
Local<>();
3.
4.      public DynamicDataSource(DataSource defaultTargetDataSource, Map<St
ring, DataSource> targetDataSources) {
5.          //设置默认数据源
6.          super.setDefaultTargetDataSource(defaultTargetDataSource);
7.          super.setTargetDataSources(new HashMap<>(targetDataSources));
8.          super.afterPropertiesSet();
9.      }
10.
11.      @Override
12.      protected Object determineCurrentLookupKey() {
13.          //获取数据源，没有指定，则为默认数据源
14.          return getDataSource();
15.      }
16.
17.      public static void setDataSource(String dataSource) {
18.          contextHolder.set(dataSource);
19.      }
20.
21.      public static String getDataSource() {
22.          return contextHolder.get();
23.      }
24.
25.      public static void clearDataSource() {
26.          contextHolder.remove();
27.      }
28.
29.  }
30.
31.
32.  public interface DataSourceNames {
33.      String FIRST = "first";

```

```

34.         String SECOND = "second";
35.
36.     }

```

- 步骤3，配置DataSource，指定数据源的信息，如下所示：

```

1.     @Configuration
2.     public class DynamicDataSourceConfig {
3.
4.         //数据源1，读取spring.datasource.druid.first下的配置信息
5.         @Bean
6.         @ConfigurationProperties("spring.datasource.druid.first")
7.         public DataSource firstDataSource() {
8.             return DruidDataSourceBuilder.create().build();
9.         }
10.
11.        //数据源2，读取spring.datasource.druid.second下的配置信息
12.        @Bean
13.        @ConfigurationProperties("spring.datasource.druid.second")
14.        public DataSource secondDataSource() {
15.            return DruidDataSourceBuilder.create().build();
16.        }
17.
18.        //加了@Primary注解，表示指定DynamicDataSource为Spring的数据源
19.        //因为DynamicDataSource是继承与AbstractRoutingDataSource，而AbstractR
        outingDataSource又是继承于AbstractDataSource，AbstractDataSource实现了统一
        的DataSource接口，所以DynamicDataSource也可以当做DataSource使用
20.        @Bean
21.        @Primary
22.        public DynamicDataSource dataSource(DataSource firstDataSource, Dat
        aSource secondDataSource) {
23.            Map<String, DataSource> targetDataSources = new HashMap<>();
24.            targetDataSources.put(DataSourceNames.FIRST, firstDataSource);
25.            targetDataSources.put(DataSourceNames.SECOND, secondDataSource)
26.            ;
27.            return new DynamicDataSource(firstDataSource, targetDataSources
28.            );
29.        }
30.    }

```

- 步骤4，通过注解，实现多数据源，如下所示：

```

1.     /**
2.      * 多数据源注解

```

```

3.     */
4.     @Target(ElementType.METHOD)
5.     @Retention(RetentionPolicy.RUNTIME)
6.     @Documented
7.     public @interface DataSource {
8.         String name() default "";
9.     }
10.
11.
12.    /**
13.     * 多数据源，切面处理类
14.     */
15.    @Aspect
16.    @Component
17.    public class DataSourceAspect implements Ordered {
18.        protected Logger logger = LoggerFactory.getLogger(getClass());
19.
20.        @Pointcut("@annotation(io.renren.datasources.annotation.DataSource)")
21.        public void dataSourcePointCut() {
22.
23.        }
24.
25.        @Around("dataSourcePointCut()")
26.        public Object around(ProceedingJoinPoint point) throws Throwable {
27.            MethodSignature signature = (MethodSignature) point.getSignature();
28.            Method method = signature.getMethod();
29.
30.            DataSource ds = method.getAnnotation(DataSource.class);
31.            if(ds == null){
32.                DynamicDataSource.setDataSource(DataSourceNames.FIRST);
33.                logger.debug("set datasource is " + DataSourceNames.FIRST);
34.            }else {
35.                DynamicDataSource.setDataSource(ds.name());
36.                logger.debug("set datasource is " + ds.name());
37.            }
38.
39.            try {
40.                return point.proceed();
41.            } finally {
42.                DynamicDataSource.clearDataSource();
43.                logger.debug("clean datasource");
44.            }
45.        }

```



```
46.
47.     @Override
48.     public int getOrder() {
49.         return 1;
50.     }
51. }
```

4.1.2. 测试多数据源

```
1.     @RunWith(SpringRunner.class)
2.     @SpringBootTest
3.     public class DynamicDataSourceTest {
4.         @Autowired
5.         private DataSourceTestService dataSourceTestService;
6.
7.         @Test
8.         public void test() {
9.             //数据源1
10.            UserEntity user = dataSourceTestService.queryObject(1L);
11.            System.out.println(ToStringBuilder.reflectionToString(user));
12.
13.            //数据源2
14.            UserEntity user2 = dataSourceTestService.queryObject2(1L);
15.            System.out.println(ToStringBuilder.reflectionToString(user2));
16.
17.            //数据源1
18.            UserEntity user3 = dataSourceTestService.queryObject(1L);
19.            System.out.println(ToStringBuilder.reflectionToString(user3));
20.        }
21.    }
22.
23.
24.
25.     @Service
26.     public class DataSourceTestService {
27.         @Autowired
28.         private UserService userService;
29.
30.         public UserEntity queryObject(Long userId) {
31.             return userService.queryObject(userId);
32.         }
33.
34.         @DataSource(name = DataSourceNames.SECOND)
```

```

35.     public UserEntity queryObject2(Long userId) {
36.         return userService.queryObject(userId);
37.     }
38. }

```

4.1.3. 增加多数据源

```

1.     spring:
2.         datasource:
3.             type: com.alibaba.druid.pool.DruidDataSource
4.             driverClassName: com.mysql.jdbc.Driver
5.             druid:
6.                 first: #数据源1
7.                     url: jdbc:mysql://192.168.1.10:3306/renren_fast?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
8.                     username: renren
9.                     password: 123456
10.                 second: #数据源2
11.                     url: jdbc:mysql://192.168.1.11:3306/order?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
12.                     username: root
13.                     password: root
14.                 third: #数据源3
15.                     url: jdbc:mysql://192.168.1.12:3306/user?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
16.                     username: root
17.                     password: root
18.
19.                 #数据源4、5、6.....

```

```

1.     public interface DataSourceNames {
2.         String FIRST = "first";
3.         String SECOND = "second";
4.         String THIRD = "third";
5.     }
6.
7.
8.     @Configuration
9.     public class DynamicDataSourceConfig {
10.
11.         @Bean
12.         @ConfigurationProperties("spring.datasource.druid.first")

```

```

13.     public DataSource firstDataSource() {
14.         return DruidDataSourceBuilder.create().build();
15.     }
16.
17.     @Bean
18.     @ConfigurationProperties("spring.datasource.druid.second")
19.     public DataSource secondDataSource() {
20.         return DruidDataSourceBuilder.create().build();
21.     }
22.
23.     @Bean
24.     @ConfigurationProperties("spring.datasource.druid.third")
25.     public DataSource thirdDataSource() {
26.         return DruidDataSourceBuilder.create().build();
27.     }
28.
29.     @Bean
30.     @Primary
31.     public DynamicDataSource dataSource(DataSource firstDataSource, DataSource secondDataSource, DataSource thirdDataSource) {
32.         Map<String, DataSource> targetDataSources = new HashMap<>();
33.         targetDataSources.put(DataSourceNames.FIRST, firstDataSource);
34.         targetDataSources.put(DataSourceNames.SECOND, secondDataSource);
35.         ;
36.         targetDataSources.put(DataSourceNames.THIRD, thirdDataSource);
37.         return new DynamicDataSource(firstDataSource, targetDataSources);
38.     }

```

4.1.4. 移除多数据源

本项目，默认是实现了多数据源的，如果自己项目不需要多数据源，也可以移除多数据源，具体操作步骤，如下所示。

步骤1，修改数据源的配置，如下所示：

```

1.     spring:
2.         datasource:
3.             type: com.alibaba.druid.pool.DruidDataSource
4.             driverClassName: com.mysql.jdbc.Driver

```

```
5.         druid:
6.             url: jdbc:mysql://localhost:3306/renren_fast?allowMultiQueries=true&useUnicode=true&characterEncoding=UTF-8
7.             username: renren
8.             password: 123456
```

步骤2，删除 `io.renren.datasources` 包下的所有类，则完成了，多数据源的移除

4.2. 核心模块

4.2.1. 前后端分离

要实现前后端分离，需要考虑以下2个问题：

1. 项目不再基于session了，如何知道访问者是谁？
2. 如何确认访问者的权限？

- 前后端分离，一般都是通过token实现，本项目也是一样；用户登录时，生成token及token过期时间，token与用户是一一对应关系，调用接口的时候，把token放到header或请求参数中，服务端就知道是谁在调用接口，登录如下所示：

```
1. //用户登录
2. @RequestMapping(value = "/sys/login", method = RequestMethod.POST)
3. public Map<String, Object> login(String username, String password) throws IOException {
4.     //用户信息
5.     SysUserEntity user = sysUserService.queryByUserName(username);
6.
7.     //账号不存在、密码错误
8.     if (user == null || !user.getPassword().equals(new Sha256Hash(password, user.getSalt()).toHex())) {
9.         return R.error("账号或密码不正确");
10.    }
11.
12.    //账号锁定
13.    if (user.getStatus() == 0) {
14.        return R.error("账号已被锁定,请联系管理员");
15.    }
16.
17.    //生成token，并保存到数据库
```

```

18.     R r = sysUserTokenService.createToken(user.getUserId());
19.     return r;
20. }
21.
22.
23. //生产token
24. public R createToken(long userId) {
25.     //生成一个token, 可以是uuid
26.     String token = TokenGenerator.generateValue();
27.
28.     //当前时间
29.     Date now = new Date();
30.     //过期时间
31.     Date expireTime = new Date(now.getTime() + EXPIRE * 1000);
32.
33.     //判断是否生成过token
34.     SysUserTokenEntity tokenEntity = queryByUserId(userId);
35.     if(tokenEntity == null){
36.         tokenEntity = new SysUserTokenEntity();
37.         tokenEntity.setUserId(userId);
38.         tokenEntity.setToken(token);
39.         tokenEntity.setUpdateTime(now);
40.         tokenEntity.setExpireTime(expireTime);
41.
42.         //保存token
43.         save(tokenEntity);
44.     }else{
45.         tokenEntity.setToken(token);
46.         tokenEntity.setUpdateTime(now);
47.         tokenEntity.setExpireTime(expireTime);
48.
49.         //更新token
50.         update(tokenEntity);
51.     }
52.
53.     R r = R.ok().put("token", token).put("expire", EXPIRE);
54.
55.     return r;
56. }

```

其中，下面的这行代码，是加盐操作；可能有人不理解为何要加盐，其目的是防止被拖库后，黑客轻易的（通过密码库对比），就能拿到你的密码

```

1.     new Sha256Hash(password, user.getSalt()).toHex()

```

- 调用接口时，接受传过来的token后，如何保证token有效及用户权限呢？其实，shiro提供了AuthenticatingFilter抽象类，继承AuthenticatingFilter抽象类即可。

步骤1，所有请求全部拒绝访问

```
1.  @Override
2.  protected boolean isAccessAllowed(ServletRequest request,
3.      ServletResponse response, Object mappedValue) {
4.      return false;
5.  }
```

步骤2，拒绝访问的请求，会调用onAccessDenied方法，onAccessDenied方法先获取token，再调用executeLogin方法

```
1.  @Override
2.  protected boolean onAccessDenied(ServletRequest request,
3.      ServletResponse response) throws Exception {
4.      //获取请求token，如果token不存在，直接返回401
5.      String token = getRequestToken((HttpServletRequest) request);
6.      if (StringUtils.isBlank(token)) {
7.          HttpServletResponse httpResponse = (HttpServletResponse) response;
8.          String json = new
9.          Gson().toJson(R.error(HttpStatus.SC_UNAUTHORIZED, "invalid token"));
10.         httpResponse.getWriter().print(json);
11.         return false;
12.     }
13.     return executeLogin(request, response);
14. }
15.
16. /**
17.  * 获取请求的token
18.  */
19. private String getRequestToken(HttpServletRequest httpRequest) {
20.     //从header中获取token
21.     String token = httpRequest.getHeader("token");
22.
23.     //如果header中不存在token，则从参数中获取token
24.     if (StringUtils.isBlank(token)) {
25.         token = httpRequest.getParameter("token");
26.     }
27. }
```

```
26.     }
27.
28.     return token;
29. }
```

步骤3，阅读AuthenticatingFilter抽象类中executeLogin方法，我们发现调用了`subject.login(token)`，这是shiro的登录方法，且需要token参数，我们自定义OAuth2Token类，只要实现AuthenticationToken接口，就可以了

```
1.  //AuthenticatingFilter类中的方法
2.  protected boolean executeLogin(ServletRequest request, ServletResponse
   response) throws Exception {
3.      AuthenticationToken token = createToken(request, response);
4.      if (token == null) {
5.          String msg = "createToken method implementation returned nu
   ll. A valid non-null AuthenticationToken " +
6.              "must be created in order to execute a login
   attempt.";
7.          throw new IllegalStateException(msg);
8.      }
9.      try {
10.         Subject subject = getSubject(request, response);
11.         subject.login(token);
12.         return onLoginSuccess(token, subject, request, response);
13.     } catch (AuthenticationException e) {
14.         return onLoginFailure(token, e, request, response);
15.     }
16. }
17.
18.
19. //OAuth2Filter类中的方法，继承了AuthenticatingFilter类
20. @Override
21. protected AuthenticationToken createToken(ServletRequest request,
   ServletResponse response) throws Exception {
22.     //获取请求token
23.     String token = getRequestToken((HttpServletRequest) request);
24.
25.     if (StringUtils.isBlank(token)) {
26.         return null;
27.     }
28.
29.     return new OAuth2Token(token);
30. }
31.
```

```

32. //subject.login(token)中的token对象, 需要实现AuthenticationToken接口
33. public class OAuth2Token implements AuthenticationToken {
34.     private String token;
35.
36.     public OAuth2Token(String token){
37.         this.token = token;
38.     }
39.
40.     @Override
41.     public String getPrincipal() {
42.         return token;
43.     }
44.
45.     @Override
46.     public Object getCredentials() {
47.         return token;
48.     }
49. }

```

步骤4, 定义OAuth2Realm类, 并继承AuthorizingRealm抽象类, 调用 `subject.login(token)` 时, 则会调用doGetAuthorizationInfo方法, 进行登录

```

1.  @Override
2.  protected AuthenticationInfo
doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
3.      String accessToken = (String) token.getPrincipal();
4.
5.      //根据accessToken, 查询用户信息
6.      SysUserTokenEntity tokenEntity =
shiroService.queryByToken(accessToken);
7.      //token失效
8.      if(tokenEntity == null || tokenEntity.getExpireTime().getTime() < S
ystem.currentTimeMillis()){
9.          throw new IncorrectCredentialsException("token失效, 请重新登录");
10.     }
11.
12.     //查询用户信息
13.     SysUserEntity user =
shiroService.queryUser(tokenEntity.getUserId());
14.     //账号锁定
15.     if(user.getStatus() == 0){
16.         throw new LockedAccountException("账号已被锁定, 请联系管理员");
17.     }

```



```

18.
19.     SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(user,
accessToken, getName());
20.     return info;
21. }

```

步骤5，登录失败后，则调用onLoginFailure，进行失败处理，整个流程结束

```

1.  @Override
2.  protected boolean onLoginFailure(AuthenticationToken token,
AuthenticationException e, ServletRequest request, ServletResponse
response) {
3.      HttpServletResponse httpResponse = (HttpServletResponse) response;
4.      httpResponse.setContentType("application/json;charset=utf-8");
5.      try {
6.          //处理登录失败的异常
7.          Throwable throwable = e.getCause() == null ? e : e.getCause();
8.          R r = R.error(HttpStatus.SC_UNAUTHORIZED, throwable.getMessage(
));
9.
10.         String json = new Gson().toJson(r);
11.         httpResponse.getWriter().print(json);
12.     } catch (IOException e1) {
13.
14.     }
15.
16.     return false;
17. }

```

步骤6，登录成功后，则调用doGetAuthorizationInfo方法，查询用户的权限，再调用具体的接口，整个流程结束

```

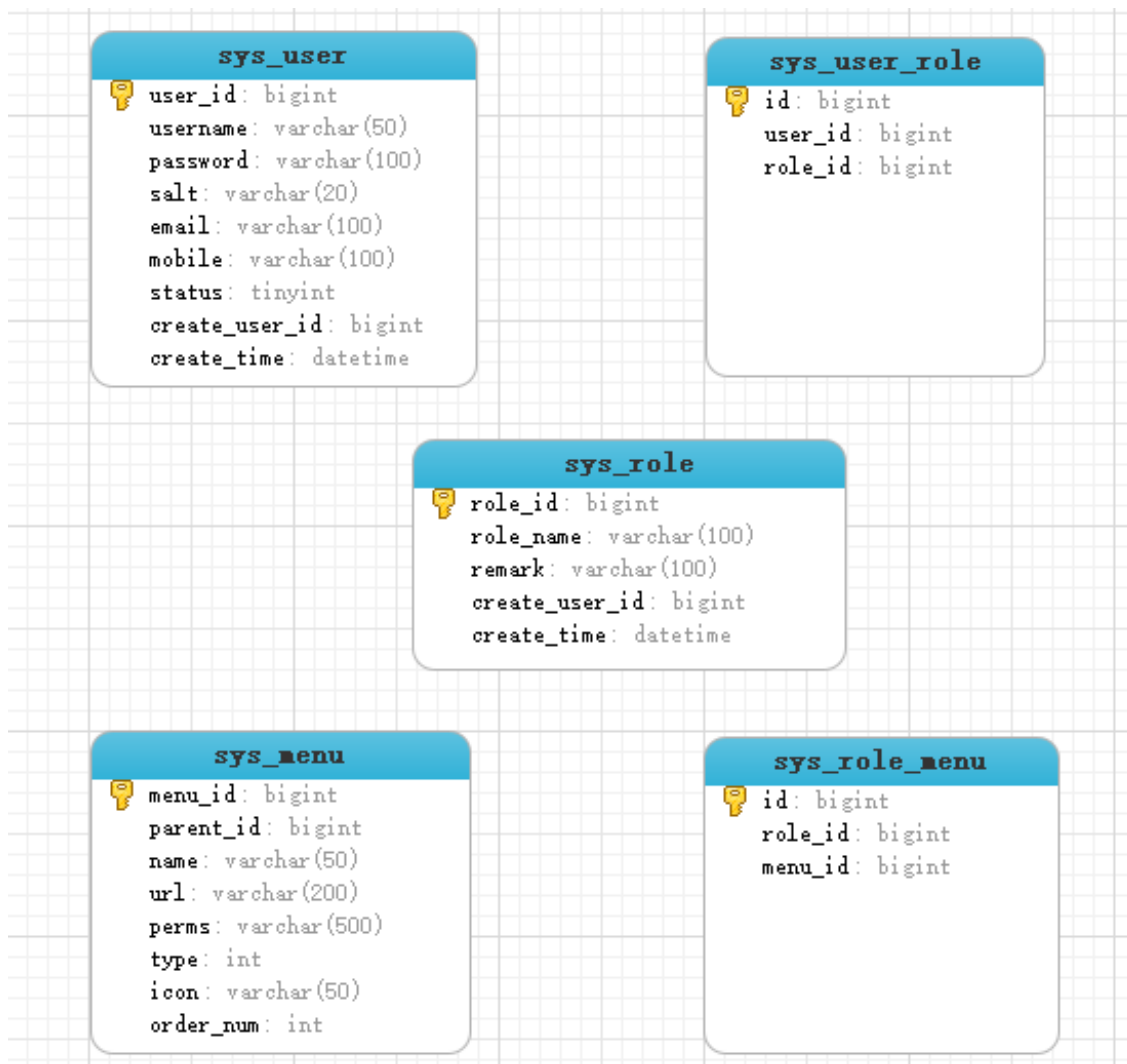
1.  @Override
2.  protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principals) {
3.      SysUserEntity user = (SysUserEntity) principals.getPrimaryPrincipal(
);
4.      Long userId = user.getUserId();
5.
6.      //用户权限列表
7.      Set<String> permsSet = shiroService.getUserPermissions(userId);
8.
9.      SimpleAuthorizationInfo info = new SimpleAuthorizationInfo();

```

```
10.     info.setStringPermissions(permsSet);
11.     return info;
12. }
```

4.2.2. 权限设计思路

权限相关的表结构，如下图所示：



1. `sys_user`[用户]表，保存用户相关数据，通过`sys_user_role`[用户与角色关联]表，与`sys_role`[角色]表关联；`sys_menu`[菜单]表通过`sys_role_menu`[菜单与角色关联]表，与`sys_role`[角色]表关联
2. `sys_menu`表，保存菜单相关数据，并在`perms`字段里，保存了shiro的权限标识，也就是说，拥有此菜单，就拥有`perms`字段里的所有权限，比如，某用户拥有的菜单权限标识 `sys:user:info`，就可以访问下面的方法

```

1.  @RequestMapping("/info/{userId}")
2.  @RequiresPermissions("sys:user:info")
3.  public R info(@PathVariable("userId") Long userId){
4.
5.  }

```

3. 在shiro配置代码里，配置为 `anon` 的，表示不经过shiro处理，配置为 `oauth2` 的，表示经过 `OAuth2Filter` 处理，前后端分离的接口，都会交给 `OAuth2Filter` 处理，这样就保证，没有权限的请求，拒绝访问

```

1.  @Configuration
2.  public class ShiroConfig {
3.
4.      @Bean("sessionManager")
5.      public SessionManager sessionManager(){
6.          DefaultWebSessionManager sessionManager = new
DefaultWebSessionManager();
7.          sessionManager.setSessionValidationSchedulerEnabled(true);
8.          sessionManager.setSessionIdCookieEnabled(false);
9.          return sessionManager;
10.     }
11.
12.     @Bean("securityManager")
13.     public SecurityManager securityManager(OAuth2Realm oAuth2Realm, Ses
sionManager sessionManager) {
14.         DefaultWebSecurityManager securityManager = new DefaultWebSecur
ityManager();
15.         securityManager.setRealm(oAuth2Realm);
16.         securityManager.setSessionManager(sessionManager);
17.
18.         return securityManager;
19.     }
20.
21.     @Bean("shiroFilter")
22.     public ShiroFilterFactoryBean shirFilter(SecurityManager
securityManager) {
23.         ShiroFilterFactoryBean shiroFilter = new ShiroFilterFactoryBean
();
24.         shiroFilter.setSecurityManager(securityManager);
25.
26.         //oauth过滤
27.         Map<String, Filter> filters = new HashMap<>();
28.         filters.put("oauth2", new OAuth2Filter());

```

```
29.         shiroFilter.setFilters(filters);
30.
31.         Map<String, String> filterMap = new LinkedHashMap<>();
32.         filterMap.put("/webjars/**", "anon");
33.         filterMap.put("/druid/**", "anon");
34.         filterMap.put("/app/**", "anon");
35.         filterMap.put("/sys/login", "anon");
36.         filterMap.put("/**/*.css", "anon");
37.         filterMap.put("/**/*.js", "anon");
38.         filterMap.put("/**/*.html", "anon");
39.         filterMap.put("/fonts/**", "anon");
40.         filterMap.put("/plugins/**", "anon");
41.         filterMap.put("/swagger/**", "anon");
42.         filterMap.put("/favicon.ico", "anon");
43.         filterMap.put("/", "anon");
44.         filterMap.put("/**", "oauth2");
45.         shiroFilter.setFilterChainDefinitionMap(filterMap);
46.
47.         return shiroFilter;
48.     }
49.
50.     @Bean("lifecycleBeanPostProcessor")
51.     public LifecycleBeanPostProcessor lifecycleBeanPostProcessor() {
52.         return new LifecycleBeanPostProcessor();
53.     }
54.
55.     @Bean
56.     public DefaultAdvisorAutoProxyCreator
57.     defaultAdvisorAutoProxyCreator() {
58.         DefaultAdvisorAutoProxyCreator proxyCreator = new DefaultAdviso
59.         rAutoProxyCreator();
60.         proxyCreator.setProxyTargetClass(true);
61.         return proxyCreator;
62.     }
63.
64.     @Bean
65.     public AuthorizationAttributeSourceAdvisor
66.     authorizationAttributeSourceAdvisor(SecurityManager securityManager) {
67.         AuthorizationAttributeSourceAdvisor advisor = new Authorization
68.         AttributeSourceAdvisor();
69.         advisor.setSecurityManager(securityManager);
70.         return advisor;
71.     }
72. }
```

4.2.3. XSS脚本过滤

XSS跨站脚本攻击的基本原理和SQL注入攻击类似，都是利用系统执行了未经过滤的危险代码，不同点在于XSS是一种基于网页脚本的注入方式，也就是将脚本攻击载荷写入网页执行以达到对网页客户端访问用户攻击的目的，属于客户端攻击。程序员往往不太关心安全这块，这就给有心之人，提供了机会，本系统针对XSS攻击，提供了过滤功能，可以有效防止XSS攻击，代码如下：

```
1.  public class XssFilter implements Filter {
2.
3.      @Override
4.      public void init(FilterConfig config) throws ServletException {
5.      }
6.
7.      public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain)
8.          throws IOException, ServletException {
9.          XssHttpServletRequestWrapper xssRequest = new
XssHttpServletRequestWrapper(
10.              (HttpServletRequest) request);
11.          chain.doFilter(xssRequest, response);
12.      }
13.
14.      @Override
15.      public void destroy() {
16.      }
17.
18.  }
19.
20.  @Configuration
21.  public class FilterConfig {
22.      @Bean
23.      public FilterRegistrationBean xssFilterRegistration() {
24.          FilterRegistrationBean registration = new
FilterRegistrationBean();
25.          registration.setDispatcherTypes(DispatcherType.REQUEST);
26.          registration.setFilter(new XssFilter());
27.          registration.addUrlPatterns("/*");
28.          registration.setName("xssFilter");
29.          registration.setOrder(Integer.MAX_VALUE);
30.          return registration;

```

```
31.     }
32. }
```

- 自定义XssFilter过滤器，用来过滤所有请求，具体过滤还是在XssHttpServletRequestWrapper里实现的，如下所示：

```
1.  public class XssHttpServletRequestWrapper extends
2.      HttpServletRequestWrapper {
3.      //没被包装过的HttpServletRequest（特殊场景，需要自己过滤）
4.      HttpServletRequest orgRequest;
5.      //html过滤
6.      private final static HTMLFilter htmlFilter = new HTMLFilter();
7.
8.      public XssHttpServletRequestWrapper(HttpServletRequest request) {
9.          super(request);
10.         orgRequest = request;
11.     }
12.
13.     @Override
14.     public ServletInputStream getInputStream() throws IOException {
15.         //非json类型，直接返回
16.         if (!MediaType.APPLICATION_JSON_VALUE.equalsIgnoreCase(super.getHeader(
17.             HttpHeaders.CONTENT_TYPE))) {
18.             return super.getInputStream();
19.         }
20.
21.         //为空，直接返回
22.         String json = IOUtils.toString(super.getInputStream(), "utf-8");
23.
24.         ;
25.         if (StringUtils.isBlank(json)) {
26.             return super.getInputStream();
27.         }
28.
29.         //xss过滤
30.         json = xssEncode(json);
31.         final ByteArrayInputStream bis = new ByteArrayInputStream(json.
32.             getBytes("utf-8"));
33.         return new ServletInputStream() {
34.             @Override
35.             public boolean isFinished() {
36.                 return true;
37.             }
38.         }
39.     }
40.
41.     @Override
42.     public String getHeader(String name) {
43.         return orgRequest.getHeader(name);
44.     }
45. }
```

```

35.         public boolean isReady() {
36.             return true;
37.         }
38.
39.         @Override
40.         public void setReadListener(ReadListener readListener) {
41.
42.         }
43.
44.         @Override
45.         public int read() throws IOException {
46.             return bis.read();
47.         }
48.     };
49. }
50.
51. @Override
52. public String getParameter(String name) {
53.     String value = super.getParameter(xssEncode(name));
54.     if (StringUtils.isNotBlank(value)) {
55.         value = xssEncode(value);
56.     }
57.     return value;
58. }
59.
60. @Override
61. public String[] getParameterValues(String name) {
62.     String[] parameters = super.getParameterValues(name);
63.     if (parameters == null || parameters.length == 0) {
64.         return null;
65.     }
66.
67.     for (int i = 0; i < parameters.length; i++) {
68.         parameters[i] = xssEncode(parameters[i]);
69.     }
70.     return parameters;
71. }
72.
73. @Override
74. public Map<String,String[]> getParameterMap() {
75.     Map<String,String[]> map = new LinkedHashMap<>();
76.     Map<String,String[]> parameters = super.getParameterMap();
77.     for (String key : parameters.keySet()) {
78.         String[] values = parameters.get(key);
79.         for (int i = 0; i < values.length; i++) {

```

```

80.         values[i] = xssEncode(values[i]);
81.     }
82.     map.put(key, values);
83. }
84. return map;
85. }
86.
87. @Override
88. public String getHeader(String name) {
89.     String value = super.getHeader(xssEncode(name));
90.     if (StringUtils.isNotBlank(value)) {
91.         value = xssEncode(value);
92.     }
93.     return value;
94. }
95.
96. private String xssEncode(String input) {
97.     return htmlFilter.filter(input);
98. }
99.
100. /**
101.  * 获取最原始的request
102.  */
103. public HttpServletRequest getOrgRequest() {
104.     return orgRequest;
105. }
106.
107. /**
108.  * 获取最原始的request
109.  */
110. public static HttpServletRequest getOrgRequest(HttpServletRequest request) {
111.     if (request instanceof XssHttpServletRequestWrapper) {
112.         return ((XssHttpServletRequestWrapper) request).getOrgRequest();
113.     }
114.
115.     return request;
116. }
117.
118. }

```

如果需要处理富文本数据，可以通

过 `XssHttpServletRequestWrapper.getOrgRequest(request)`，拿到原始的 `request` 对象后，

4.2.4. SQL注入

本系统使用的是Mybatis，如果使用\${}拼接SQL，则存在SQL注入风险，可以对参数进行过滤，避免SQL注入，如下：

```
1.  public class SQLFilter {
2.
3.      /**
4.       * SQL注入过滤
5.       * @param str 待验证的字符串
6.       */
7.      public static String sqlInject(String str){
8.          if(StringUtils.isBlank(str)){
9.              return null;
10.         }
11.         //去掉'|"|\;字符
12.         str = StringUtils.replace(str, "'", "");
13.         str = StringUtils.replace(str, "\"", "");
14.         str = StringUtils.replace(str, ";", "");
15.         str = StringUtils.replace(str, "\\\"", "");
16.
17.         //转换成小写
18.         str = str.toLowerCase();
19.
20.         //非法字符
21.         String[] keywords = {"master", "truncate", "insert", "select",
22. "delete", "update", "declare", "alert", "drop"};
23.
24.         //判断是否包含非法字符
25.         for(String keyword : keywords){
26.             if(str.indexOf(keyword) != -1){
27.                 throw new RuntimeException("包含非法字符");
28.             }
29.         }
30.         return str;
31.     }
32. }
```

像查询列表，涉及排序问题，排序字段是从前台传过来的，则存在SQL注入风险，需经如下处理：

```
1. public class Query extends LinkedHashMap<String, Object> {
2.     private static final long serialVersionUID = 1L;
3.     //当前页码
4.     private int page;
5.     //每页条数
6.     private int limit;
7.
8.     public Query(Map<String, Object> params) {
9.         this.putAll(params);
10.
11.         //分页参数
12.         this.page = Integer.parseInt(params.get("page").toString());
13.         this.limit = Integer.parseInt(params.get("limit").toString());
14.         this.put("offset", (page - 1) * limit);
15.         this.put("page", page);
16.         this.put("limit", limit);
17.
18.         //防止SQL注入（因为sidx、order是通过拼接SQL实现排序的，会有SQL注入风险）
19.         String idx = (String)params.get("sidx");
20.         String order = (String)params.get("order");
21.         if(StringUtils.isNotBlank(idx)){
22.             this.put("sidx", SQLFilter.sqlInject(idx));
23.         }
24.         if(StringUtils.isNotBlank(order)){
25.             this.put("order", SQLFilter.sqlInject(order));
26.         }
27.
28.     }
29. }
```

4.2.5. Redis缓存

缓存大家都很熟悉，但能否灵活运用，就不一定了。一般设计缓存架构时，我们需要考虑如下几个问题：

1. 查询数据的时候，尽量减少DB查询，DB主要负责写数据
2. 尽量不使用 `LEFT JOIN` 等关联查询，缓存命中率不高，还浪费内存
3. 多使用单表查询，缓存命中率最高
4. 数据库 `insert`、`update`、`delete` 时，同步更新缓存数据

5. 合理运用Redis数据结构，也许有质的飞跃
6. 对于访问量不大的项目，使用缓存只会增加项目的复杂度

本系统采用Redis作为缓存，并可配置是否开启redis缓存，主要还是通过Spring AOP实现的，配置如下所示：

```
1.  redis:
2.      open: false  # 是否开启redis缓存  true开启  false关闭
3.      database: 0
4.      host: redis.open.renren.io
5.      port: 16379
6.      password:      # 密码（默认为空）
7.      timeout: 6000  # 连接超时时长（毫秒）
8.      pool:
9.          max-active: 1000  # 连接池最大连接数（使用负值表示没有限制）
10.         max-wait: -1      # 连接池最大阻塞等待时间（使用负值表示没有限制）
11.         max-idle: 10      # 连接池中的最大空闲连接
12.         min-idle: 5       # 连接池中的最小空闲连接
```

本项目中，使用Redis服务的代码，如下所示：

```
1.  public class SysConfigServiceImpl implements SysConfigService {
2.      @Autowired
3.      private SysConfigDao sysConfigDao;
4.      @Autowired
5.      private SysConfigRedis sysConfigRedis;
6.
7.      @Override
8.      @Transactional
9.      public void save(SysConfigEntity config) {
10.         sysConfigDao.save(config);
11.         sysConfigRedis.saveOrUpdate(config);
12.     }
13.
14.     @Override
15.     @Transactional
16.     public void update(SysConfigEntity config) {
17.         sysConfigDao.update(config);
18.         sysConfigRedis.saveOrUpdate(config);
19.     }
20.
21.     @Override
22.     @Transactional
```

```

23.     public void updateValueByKey(String key, String value) {
24.         sysConfigDao.updateValueByKey(key, value);
25.         sysConfigRedis.delete(key);
26.     }
27.
28.     @Override
29.     @Transactional
30.     public void deleteBatch(Long[] ids) {
31.         sysConfigDao.deleteBatch(ids);
32.
33.         for(Long id : ids){
34.             SysConfigEntity config = queryObject(id);
35.             sysConfigRedis.delete(config.getKey());
36.         }
37.     }
38. }
39.
40.
41. -----
42.
43.
44. @Component
45. public class SysConfigRedis {
46.     @Autowired
47.     private RedisUtils redisUtils;
48.
49.     public void saveOrUpdate(SysConfigEntity config) {
50.         if(config == null){
51.             return ;
52.         }
53.         String key = RedisKeys.getSysConfigKey(config.getKey());
54.         redisUtils.set(key, config);
55.     }
56.
57.     public void delete(String configKey) {
58.         String key = RedisKeys.getSysConfigKey(configKey);
59.         redisUtils.delete(key);
60.     }
61.
62.     public SysConfigEntity get(String configKey){
63.         String key = RedisKeys.getSysConfigKey(configKey);
64.         return redisUtils.get(key, SysConfigEntity.class);
65.     }
66. }
67.

```

```
68. -----
69.
70.
71.
72. package io.renren.common.aspect;
73.
74. @Component
75. public class RedisUtils {
76.     @Autowired
77.     private RedisTemplate<String, Object> redisTemplate;
78.     @Autowired
79.     private ValueOperations<String, String> valueOperations;
80.     @Autowired
81.     private HashOperations<String, String, Object> hashOperations;
82.     @Autowired
83.     private ListOperations<String, Object> listOperations;
84.     @Autowired
85.     private SetOperations<String, Object> setOperations;
86.     @Autowired
87.     private ZSetOperations<String, Object> zSetOperations;
88.     /** 默认过期时长, 单位: 秒 */
89.     public final static long DEFAULT_EXPIRE = 60 * 60 * 24;
90.     /** 不设置过期时长 */
91.     public final static long NOT_EXPIRE = -1;
92.     private final static Gson gson = new Gson();
93.
94.     public void set(String key, Object value, long expire){
95.         valueOperations.set(key, toJson(value));
96.         if(expire != NOT_EXPIRE){
97.             redisTemplate.expire(key, expire, TimeUnit.SECONDS);
98.         }
99.     }
100.
101.     public void set(String key, Object value){
102.         set(key, value, DEFAULT_EXPIRE);
103.     }
104.
105.     public <T> T get(String key, Class<T> clazz, long expire) {
106.         String value = valueOperations.get(key);
107.         if(expire != NOT_EXPIRE){
108.             redisTemplate.expire(key, expire, TimeUnit.SECONDS);
109.         }
110.         return value == null ? null : fromJson(value, clazz);
111.     }
112.
```

```

113.     public <T> T get(String key, Class<T> clazz) {
114.         return get(key, clazz, NOT_EXPIRE);
115.     }
116.
117.     public String get(String key, long expire) {
118.         String value = valueOperations.get(key);
119.         if(expire != NOT_EXPIRE){
120.             redisTemplate.expire(key, expire, TimeUnit.SECONDS);
121.         }
122.         return value;
123.     }
124.
125.     public String get(String key) {
126.         return get(key, NOT_EXPIRE);
127.     }
128.
129.     public void delete(String key) {
130.         redisTemplate.delete(key);
131.     }
132.
133.     /**
134.      * Object转成JSON数据
135.      */
136.     private String toJson(Object object){
137.         if(object instanceof Integer || object instanceof Long || object
138.         t instanceof Float ||
139.         object instanceof Double || object instanceof Boolean |
140.         | object instanceof String){
141.             return String.valueOf(object);
142.         }
143.         return gson.toJson(object);
144.     }
145.
146.     /**
147.      * JSON数据, 转成Object
148.      */
149.     private <T> T fromJson(String json, Class<T> clazz){
150.         return gson.fromJson(json, clazz);
151.     }

```

大家可能会有疑问，认为这个项目必须要配置Redis缓存，否则会报错，因为有操作Redis的代码，其实不然，通过Spring AOP，我们可以控制，是否真的使用Redis，代码如下：

```

1.  @Aspect
2.  @Configuration
3.  public class RedisAspect {
4.      private Logger logger = LoggerFactory.getLogger(getClass());
5.      //是否开启redis缓存 true开启 false关闭
6.      @Value("${spring.redis.open: false}")
7.      private boolean open;
8.
9.      @Around("execution(* io.renren.common.utils.RedisUtils.*(..))")
10.     public Object around(ProceedingJoinPoint point) throws Throwable {
11.         Object result = null;
12.         if (open) {
13.             try {
14.                 result = point.proceed();
15.             } catch (Exception e) {
16.                 logger.error("redis error", e);
17.                 throw new RRException("Redis服务异常");
18.             }
19.         }
20.         return result;
21.     }
22. }

```

4.2.6. 异常处理机制

本项目通过RRException异常类，抛出自定义异常，RRException继承RuntimeException，不能继承Exception，如果继承Exception，则Spring事务不会回滚。

RRException代码如下所示：

```

1.  public class RRException extends RuntimeException {
2.      private static final long serialVersionUID = 1L;
3.
4.      private String msg;
5.      private int code = 500;
6.
7.      public RRException(String msg) {
8.          super(msg);
9.          this.msg = msg;

```

```

10.     }
11.
12.     public RRException(String msg, Throwable e) {
13.         super(msg, e);
14.         this.msg = msg;
15.     }
16.
17.     public RRException(String msg, int code) {
18.         super(msg);
19.         this.msg = msg;
20.         this.code = code;
21.     }
22.
23.     public RRException(String msg, int code, Throwable e) {
24.         super(msg, e);
25.         this.msg = msg;
26.         this.code = code;
27.     }
28.
29.     public String getMsg() {
30.         return msg;
31.     }
32.
33.     public void setMsg(String msg) {
34.         this.msg = msg;
35.     }
36.
37.     public int getCode() {
38.         return code;
39.     }
40.
41.     public void setCode(int code) {
42.         this.code = code;
43.     }

```

如何处理抛出的异常呢，我们定义了RRExceptionHandler类，并加上注解@RestControllerAdvice，就可以处理所有抛出的异常，并返回JSON数据。@RestControllerAdvice是由@ControllerAdvice、@ResponseBody注解组合而来的，可以查找@ControllerAdvice相关的资料，理解@ControllerAdvice注解的使用。

RRExceptionHandler代码如下所示：


```

1.  @RestControllerAdvice
2.  public class RREExceptionHandler {
3.      private Logger logger = LoggerFactory.getLogger(getClass());
4.
5.      /**
6.       * 处理自定义异常
7.       */
8.      @ExceptionHandler(RREException.class)
9.      public R handleRREException(RREException e) {
10.         R r = new R();
11.         r.put("code", e.getCode());
12.         r.put("msg", e.getMessage());
13.
14.         return r;
15.     }
16.
17.     @ExceptionHandler(DuplicateKeyException.class)
18.     public R handleDuplicateKeyException(DuplicateKeyException e) {
19.         logger.error(e.getMessage(), e);
20.         return R.error("数据库中已存在该记录");
21.     }
22.
23.     @ExceptionHandler(AuthorizationException.class)
24.     public R handleAuthorizationException(AuthorizationException e) {
25.         logger.error(e.getMessage(), e);
26.         return R.error("没有权限，请联系管理员授权");
27.     }
28.
29.     @ExceptionHandler(Exception.class)
30.     public R handleException(Exception e) {
31.         logger.error(e.getMessage(), e);
32.         return R.error();
33.     }
34. }

```

4.2.7. 后端效验机制

本项目，后端效验使用的是Hibernate Validator校验框架，且自定义ValidatorUtils工具类，用来效验数据。

Hibernate Validator官方文档：

http://docs.jboss.org/hibernate/validator/5.4/reference/en-US/html_single/

ValidatorUtils代码如下所示：

```
1.  public class ValidatorUtils {
2.      private static Validator validator;
3.
4.      static {
5.          validator = Validation.buildDefaultValidatorFactory().getValida
tor();
6.      }
7.
8.      /**
9.       * 校验对象
10.      * @param object      待校验对象
11.      * @param groups      待校验的组
12.      * @throws RRException 校验不通过，则报RRException异常
13.      */
14.      public static void validateEntity(Object object, Class<?>... groups
)
15.          throws RRException {
16.          Set<ConstraintViolation<Object>> constraintViolations = validat
or.validate(object, groups);
17.          if (!constraintViolations.isEmpty()) {
18.              StringBuilder msg = new StringBuilder();
19.              for (ConstraintViolation<Object> constraint:
constraintViolations) {
20.                  msg.append(constraint.getMessage()).append("<br>");
21.              }
22.              throw new RRException(msg.toString());
23.          }
24.      }
25.  }
```

使用案例：

```
1.  @RestController
2.  @RequestMapping("/sys/user")
3.  public class SysUserController extends AbstractController {
4.      /**
5.       * 保存用户
6.       */
7.      @SysLog("保存用户")
```

```

8.     @RequestMapping("/save")
9.     @RequiresPermissions("sys:user:save")
10.    public R save(@RequestBody SysUserEntity user){
11.        //保存用户时, 效验SysUserEntity里, 带有AddGroup注解的属性
12.        ValidatorUtils.validateEntity(user, AddGroup.class);
13.
14.        user.setCreateUserId(getUserId());
15.        sysUserService.save(user);
16.
17.        return R.ok();
18.    }
19.
20.    /**
21.     * 修改用户
22.     */
23.    @SysLog("修改用户")
24.    @RequestMapping("/update")
25.    @RequiresPermissions("sys:user:update")
26.    public R update(@RequestBody SysUserEntity user){
27.        //修改用户时, 效验SysUserEntity里, 带有UpdateGroup注解的属性
28.        ValidatorUtils.validateEntity(user, UpdateGroup.class);
29.
30.        user.setCreateUserId(getUserId());
31.        sysUserService.update(user);
32.
33.        return R.ok();
34.    }
35. }
36.
37.
38. -----
39.
40.
41. public class SysUserEntity implements Serializable {
42.     /**
43.      * 用户ID
44.      */
45.     private Long userId;
46.
47.     /**
48.      * 用户名
49.      */
50.     @NotBlank(message="用户名不能为空", groups = {AddGroup.class, UpdateG
51.         roup.class})
        private String username;

```

```
52.
53.     /**
54.      * 密码
55.      */
56.     @NotBlank(message="密码不能为空", groups = AddGroup.class)
57.     private String password;
58.
59.     /**
60.      * 盐
61.      */
62.     private String salt;
63.
64.     /**
65.      * 邮箱
66.      */
67.     @NotBlank(message="邮箱不能为空", groups = {AddGroup.class, UpdateGro
68. up.class})
69.     @Email(message="邮箱格式不正确", groups = {AddGroup.class,
70. UpdateGroup.class})
71.     private String email;
72.
73.     /**
74.      * 手机号
75.      */
76.     private String mobile;
77.
78.     /**
79.      * 状态 0：禁用 1：正常
80.      */
81.     private Integer status;
82.
83.     /**
84.      * 角色ID列表
85.      */
86.     private List<Long> roleIdList;
87.
88.     /**
89.      * 创建者ID
90.      */
91.     private Long createUserId;
92.
93.     /**
94.      * 创建时间
95.      */
96.     private Date createTime;
```

```
95.     }
```

通过分析上面的代码，我们来理解Hibernate Validator校验框架的使用。

其中，username属性，表示保存或修改用户时，都会校验username属性；而password属性，表示只有保存用户时，才会校验password属性，也就是说，修改用户时，password可以不填写，允许为空。如果不指定属性的groups，则默认属于javax.validation.groups.Default.class分组，可以通过ValidatorUtils.validateEntity(user)进行校验。

4.2.8. 系统日志

系统日志是通过Spring AOP实现的，我们自定义了注解@SysLog，且只能在方法上使用，如下所示：

```
1.     @Target(ElementType.METHOD)
2.     @Retention(RetentionPolicy.RUNTIME)
3.     @Documented
4.     public @interface SysLog {
5.
6.         String value() default "";
7.     }
```

下面是自定义注解@SysLog的使用方式，如下所示：

```
1.     @RestController
2.     @RequestMapping("/sys/user")
3.     public class SysUserController extends AbstractController {
4.
5.         @SysLog("保存用户")
6.         @RequestMapping("/save")
7.         @RequiresPermissions("sys:user:save")
8.         public R save(@RequestBody SysUserEntity user) {
9.             ValidatorUtils.validateEntity(user, AddGroup.class);
10.
11.             user.setCreateUserId(getUserId());
12.             sysUserService.save(user);
13.
14.             return R.ok();
15.         }
16.     }
```

我们可以发现，只需要在保存日志的请求方法上，加上 `@SysLog` 注解，就可以把日志保存到数据库里了。

具体是在哪里把数据保存到数据库里的呢，我们定义了 `SysLogAspect` 处理类，就是来干这事的，如下所示：

```
1.  /**
2.   * 系统日志，切面处理类
3.   *
4.   * @author chenshun
5.   * @email sunlightcs@gmail.com
6.   * @date 2017年3月8日 上午11:07:35
7.   */
8.  @Aspect
9.  @Component
10. public class SysLogAspect {
11.     @Autowired
12.     private SysLogService sysLogService;
13.
14.     @Pointcut("@annotation(io.renren.common.annotation.SysLog)")
15.     public void logPointCut() {
16.
17.     }
18.
19.     @Around("logPointCut()")
20.     public Object around(ProceedingJoinPoint point) throws Throwable {
21.         long beginTime = System.currentTimeMillis();
22.         //执行方法
23.         Object result = point.proceed();
24.         //执行时长(毫秒)
25.         long time = System.currentTimeMillis() - beginTime;
26.
27.         //保存日志
28.         saveSysLog(point, time);
29.
30.         return result;
31.     }
32.
33.     private void saveSysLog(ProceedingJoinPoint joinPoint, long time) {
34.         MethodSignature signature = (MethodSignature) joinPoint.getSignature();
35.         Method method = signature.getMethod();
36.
37.         SysLogEntity sysLog = new SysLogEntity();
38.         SysLog syslog = method.getAnnotation(SysLog.class);
```

```

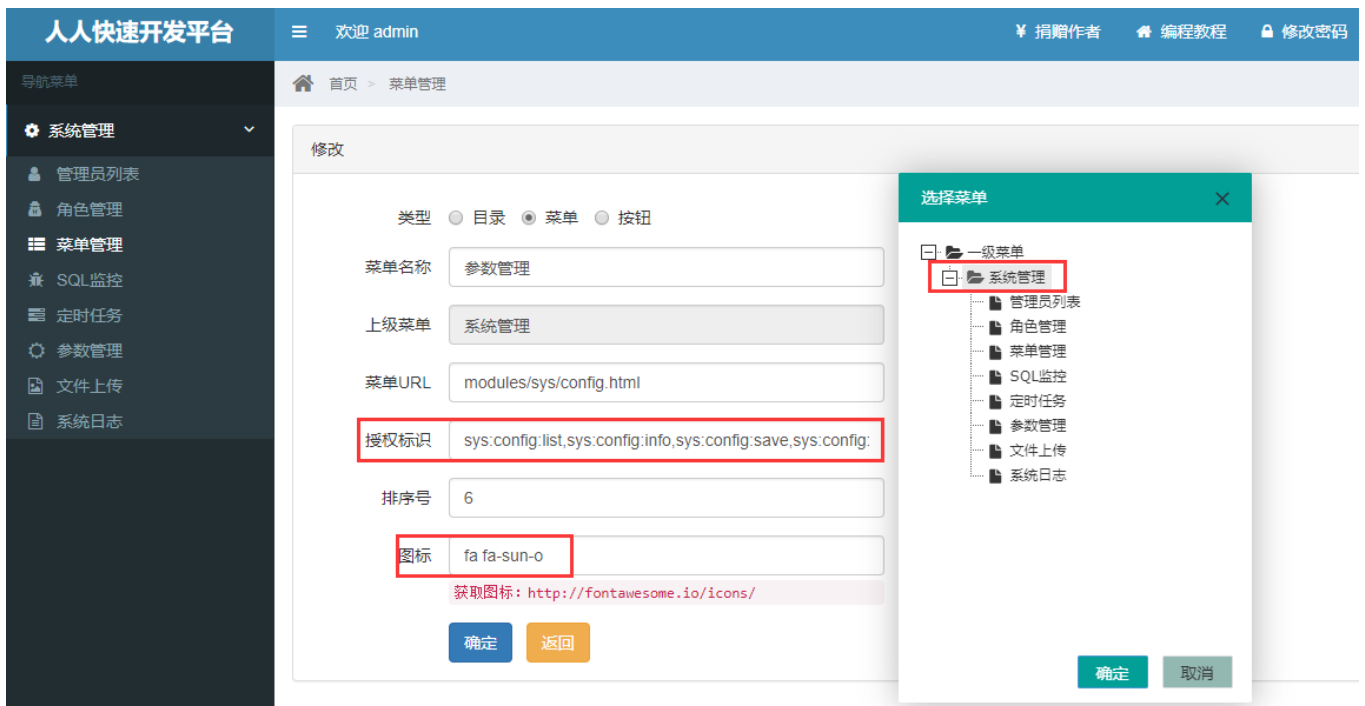
39.         if(syslog != null){
40.             //注解上的描述
41.             syslog.setOperation(syslog.value());
42.         }
43.
44.         //请求的方法名
45.         String className = joinPoint.getTarget().getClass().getName();
46.         String methodName = signature.getName();
47.         syslog.setMethod(className + "." + methodName + "()");
48.
49.         //请求的参数
50.         Object[] args = joinPoint.getArgs();
51.         try{
52.             String params = new Gson().toJson(args[0]);
53.             syslog.setParams(params);
54.         }catch (Exception e){
55.
56.         }
57.
58.         //获取request
59.         HttpServletRequest request =
HttpContextUtils.getHttpServletRequest();
60.         //设置IP地址
61.         syslog.setIp(IPUtils.getIpAddr(request));
62.
63.         //用户名
64.         String username = ((SysUserEntity) SecurityUtils.getSubject().getPrincipal()).getUsername();
65.         syslog.setUsername(username);
66.
67.         syslog.setTime(time);
68.         syslog.setCreateDate(new Date());
69.         //保存系统日志
70.         syslogService.save(syslog);
71.     }
72. }

```

`SysLogAspect` 类定义了一个切入点，请求 `@SysLog` 注解的方法时，会进入 `around` 方法，把系统日志保存到数据库中。

4.2.9. 添加菜单

菜单管理，主要是对【目录、菜单、按钮】进行动态的新增、修改、删除等操作，方便开发者管理菜单。



上图是拿现有的菜单进行讲解。其中，授权标识与shiro中的注解@RequiresPermissions，定义的授权标识是一一对应的，如下所示：

```
1. @RestController
2. @RequestMapping("/sys/config")
3. public class SysConfigController extends AbstractController {
4.
5.     @RequestMapping("/list")
6.     @RequiresPermissions("sys:config:list")
7.     public R list(@RequestParam Map<String, Object> params) {
8.
9.     }
10.
11.    @RequestMapping("/info/{id}")
12.    @RequiresPermissions("sys:config:info")
13.    public R info(@PathVariable("id") Long id) {
14.
15.    }
16.
17.    @RequestMapping("/save")
18.    @RequiresPermissions("sys:config:save")
19.    public R save(@RequestBody SysConfigEntity config) {
```



```

20.
21.     }
22.
23.     @RequestMapping("/update")
24.     @RequiresPermissions("sys:config:update")
25.     public R update(@RequestBody SysConfigEntity config){
26.
27.     }
28.
29.     @RequestMapping("/delete")
30.     @RequiresPermissions("sys:config:delete")
31.     public R delete(@RequestBody Long[] ids){
32.
33.     }
34.
35. }

```

4.2.10. 添加角色

管理员权限是通过角色进行管理的，给管理员分配权限时，要先创建好角色。

下面创建了一个【开发角色】，如下图所示：

人人快速开发平台

欢迎 admin

捐赠作者 编程教程 修改密码

导航菜单

- 系统管理
 - 管理员列表
 - 角色管理
 - 菜单管理
 - SQL监控
 - 定时任务
 - 参数管理
 - 文件上传
 - 系统日志

首页 > 角色管理

新增

角色名称 开发角色

备注 开发人员使用

授权

- ☒ 系统管理
 - ☒ 管理员列表
 - ☒ 角色管理
 - ☒ 菜单管理
 - ☒ SQL监控
 - ☒ 定时任务
 - ☒ 参数管理
 - ☒ 文件上传
 - ☒ 系统日志

确定 返回

4.2.11. 添加管理员

本系统默认就创建了admin账号，无需分配任何角色，就拥有最高权限。
一个管理员是可以拥有多个角色的。
非admin账号，只能查看及管理自己创建的账号。

下面创建一个【zhangsan】的管理员账号，并属于【开发角色】，如下所示：

人人快速开发平台 欢迎 admin 捐赠作者 编程教程 修改密码

导航菜单

- 系统管理
- 管理员列表
- 角色管理
- 菜单管理
- SQL监控
- 定时任务
- 参数管理
- 文件上传
- 系统日志

首页 > 管理员列表

新增

用户名 zhangsan

密码 123456

邮箱 zhangsan@renren.io

手机号 13512345678

角色 ☒ 开发角色

状态 ☐ 禁用 ☒ 正常

确定 返回

4.3. 定时任务模块

本系统使用开源框架Quartz，实现的定时任务，已实现分布式定时任务，可部署多台服务器，不重复执行，以及动态增加、修改、删除、暂停、恢复、立即执行定时任务。Quartz自带了各数据库的SQL脚本，如果想更改成其他数据库，可参考Quartz相应的SQL脚本。

4.3.1. 新增定时任务

新增一个定时任务，其实很简单，只要定义一个普通的Spring Bean即可，如下所示：

```
1. @Component("testTask")
2. public class TestTask {
3.     private Logger logger = LoggerFactory.getLogger(getClass());
4.
5.     @Autowired
```

```

6.     private SysUserService sysUserService;
7.
8.     //定时任务只能接受一个参数；如果有多个参数，使用json数据即可
9.     public void test(String params){
10.         logger.info("我是带参数的test方法，正在被执行，参数为：" + params);
11.
12.         try {
13.             Thread.sleep(1000L);
14.         } catch (InterruptedException e) {
15.             e.printStackTrace();
16.         }
17.
18.         SysUserEntity user = sysUserService.queryObject(1L);
19.         System.out.println(ToStringBuilder.reflectionToString(user));
20.
21.     }
22.
23.
24.     public void test2(){
25.         logger.info("我是不带参数的test2方法，正在被执行");
26.     }
27. }

```

如何让Quartz，定时执行testTask里的方法呢？只需要在管理后台，新增一个定时任务即可，如下图所示：

人人快速开发平台

≡ 欢迎 admin

导航菜单

⚙️ 系统管理

👤 管理员列表

👤 角色管理

☰ 菜单管理

🔍 SQL监控

📅 定时任务

⚙️ 参数管理

📁 文件上传

📖 系统日志

🏠 首页 > 定时任务

新增

bean名称

testTask

方法名称

test

参数

renren

cron表达式

* / 10 * * * ?

备注

备注

确定

返回

首页 > 定时任务

bean名称

查询

+ 新增

修改

删除

暂停

恢复

立即执行

日志列表

	<input type="checkbox"/>	任务ID	bean名称	方法名称	参数	cron表达式	备注	状态
1	<input type="checkbox"/>	61	testTask	test	renren	* / 10 * * * ?		正常

刚才配置的定时任务，每隔10秒，就会调用TestTask的test方法了，是不是很简单啊。

4.3.2. 源码分析

Quartz提供了相关的API，我们可以调用API，对Quartz进行增加、修改、删除、暂停、恢复、立即执行等。本系统中，`ScheduleUtils`类就是对Quartz API进行的封装，代码如下所示：

```
1. public class ScheduleUtils {
2.     private final static String JOB_NAME = "TASK_";
3.
4.     /**
5.      * 获取触发器key
6.      */
7.     private static TriggerKey getTriggerKey(Long jobId) {
8.         return TriggerKey.triggerKey(JOB_NAME + jobId);
9.     }
10.
11.    /**
12.     * 获取jobKey
13.     */
14.    private static JobKey getJobKey(Long jobId) {
15.        return JobKey.jobKey(JOB_NAME + jobId);
16.    }
17.
18.    /**
19.     * 获取表达式触发器
20.     */
21.    public static CronTrigger getCronTrigger(Scheduler scheduler, Long
jobId) {
22.        try {
23.            return (CronTrigger) scheduler.getTrigger(getTriggerKey(job
Id));
24.        } catch (Exception e) {
25.            // TODO: handle exception
26.        }
27.    }
28. }
```

```

24.         } catch (SchedulerException e) {
25.             throw new RRException("getCronTrigger异常, 请检查qrtz开头的表
, 是否有脏数据", e);
26.         }
27.     }
28.
29.     /**
30.      * 创建定时任务
31.      */
32.     public static void createScheduleJob(Scheduler scheduler,
ScheduleJobEntity scheduleJob) {
33.         try {
34.             //构建job信息
35.             JobDetail jobDetail = JobBuilder.newJob(ScheduleJob.class).
withIdentity(getJobKey(scheduleJob.getJobId())).build();
36.
37.             //表达式调度构建器
38.             CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.c
ronSchedule(scheduleJob.getCronExpression())
39.                 .withMisfireHandlingInstructionDoNothing();
40.
41.             //按新的cronExpression表达式构建一个新的trigger
42.             CronTrigger trigger = TriggerBuilder.newTrigger().withIdent
ity(getTriggerKey(scheduleJob.getJobId())).
43.                 withSchedule(scheduleBuilder).build();
44.
45.             //放入参数，运行时的方法可以获取
46.             jobDetail.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_K
EY, new Gson().toJson(scheduleJob));
47.
48.             scheduler.scheduleJob(jobDetail, trigger);
49.
50.             //暂停任务
51.             if(scheduleJob.getStatus() == ScheduleStatus.PAUSE.getValue
()){
52.                 pauseJob(scheduler, scheduleJob.getJobId());
53.             }
54.         } catch (SchedulerException e) {
55.             throw new RRException("创建定时任务失败", e);
56.         }
57.     }
58.
59.     /**
60.      * 更新定时任务
61.      */

```

```

62.     public static void updateScheduleJob(Scheduler scheduler,
ScheduleJobEntity scheduleJob) {
63.         try {
64.             TriggerKey triggerKey = getTriggerKey(scheduleJob.getJobId(
));
65.
66.             //表达式调度构建器
67.             CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.c
ronSchedule(scheduleJob.getCronExpression())
68.                 .withMisfireHandlingInstructionDoNothing();
69.
70.             CronTrigger trigger = getCronTrigger(scheduler, scheduleJob
.getJobId());
71.
72.             //按新的cronExpression表达式重新构建trigger
73.             trigger = trigger.getTriggerBuilder().withIdentity(triggerK
ey).withSchedule(scheduleBuilder).build();
74.
75.             //参数
76.             trigger.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY
, new Gson().toJson(scheduleJob));
77.
78.             scheduler.rescheduleJob(triggerKey, trigger);
79.
80.             //暂停任务
81.             if(scheduleJob.getStatus() == ScheduleStatus.PAUSE.getValue
()){
82.                 pauseJob(scheduler, scheduleJob.getJobId());
83.             }
84.
85.         } catch (SchedulerException e) {
86.             throw new RuntimeException("更新定时任务失败", e);
87.         }
88.     }
89.
90.     /**
91.      * 立即执行任务
92.      */
93.     public static void run(Scheduler scheduler, ScheduleJobEntity
scheduleJob) {
94.         try {
95.             //参数
96.             JobDataMap dataMap = new JobDataMap();
97.             dataMap.put(ScheduleJobEntity.JOB_PARAM_KEY, new Gson().toJ
son(scheduleJob));

```

```

98.
99.         scheduler.triggerJob(getJobKey(scheduleJob.getJobId()), dat
aMap);
100.     } catch (SchedulerException e) {
101.         throw new RRException("立即执行定时任务失败", e);
102.     }
103. }
104.
105. /**
106.  * 暂停任务
107.  */
108. public static void pauseJob(Scheduler scheduler, Long jobId) {
109.     try {
110.         scheduler.pauseJob(getJobKey(jobId));
111.     } catch (SchedulerException e) {
112.         throw new RRException("暂停定时任务失败", e);
113.     }
114. }
115.
116. /**
117.  * 恢复任务
118.  */
119. public static void resumeJob(Scheduler scheduler, Long jobId) {
120.     try {
121.         scheduler.resumeJob(getJobKey(jobId));
122.     } catch (SchedulerException e) {
123.         throw new RRException("暂停定时任务失败", e);
124.     }
125. }
126.
127. /**
128.  * 删除定时任务
129.  */
130. public static void deleteScheduleJob(Scheduler scheduler, Long
jobId) {
131.     try {
132.         scheduler.deleteJob(getJobKey(jobId));
133.     } catch (SchedulerException e) {
134.         throw new RRException("删除定时任务失败", e);
135.     }
136. }
137. }

```

以下是几个核心的方法：

- createScheduleJob【创建定时任务】：在管理后台新增任务时，会调用该方法，把任务添加到Quartz中，再根据cron表达式，定时执行任务。
- updateScheduleJob【更新定时任务】：修改任务时，调用该方法，修改Quartz中的任务信息。
- run【立即执行定时任务】：马上执行一次该任务，只执行一次。
- pauseJob【暂停定时任务】：这个不是暂停正在执行的任务，而是以后不再执行这个定时任务了。正在执行的任务，还是照常执行完。
- resumeJob【恢复定时任务】：这个是针对pauseJob来的，如果任务暂停了，以后都不会再执行，要想再执行，则需要调用resumeJob，使定时任务恢复执行。
- deleteScheduleJob【删除定时任务】：删除定时任务

其中，createScheduleJob、updateScheduleJob在启动项目的时候，也会调用，把数据库里，新增或修改的任务，更新到Quartz中，如下所示：

```
1.  @Service("scheduleJobService")
2.  public class ScheduleJobServiceImpl implements ScheduleJobService {
3.      /**
4.       * 项目启动时，初始化定时器
5.       */
6.      @PostConstruct
7.      public void init(){
8.          List<ScheduleJobEntity> scheduleJobList = schedulerJobDao.query
List(new HashMap<>());
9.          for(ScheduleJobEntity scheduleJob : scheduleJobList){
10.              CronTrigger cronTrigger = ScheduleUtils.getCronTrigger(sche
duler, scheduleJob.getJobId());
11.              //如果不存在，则创建
12.              if(cronTrigger == null) {
13.                  ScheduleUtils.createScheduleJob(scheduler, scheduleJob)
;
14.              }else {
15.                  ScheduleUtils.updateScheduleJob(scheduler, scheduleJob)
;
16.              }
17.          }
18.      }
19.
20.  }
```

大家是不是还有疑问呢，怎么就能定时执行，刚才在管理后台新增的任务testTask呢？

下面我们再来分析下createScheduleJob方法，创建定时任务的时候，要调用该方法，代码如下所示：


```

1. //构建一个新的定时任务, JobBuilder.newJob() 只能接受Job类型的参数
2. //把ScheduleJob.class作为参数传进去, ScheduleJob继承QuartzJobBean, 而
   QuartzJobBean实现了Job接口
3. JobDetail jobDetail =
   JobBuilder.newJob(ScheduleJob.class).withIdentity(getJobKey(scheduleJob
   .getJobId())).build();
4.
5. //构建cron, 定时任务的周期
6. CronScheduleBuilder scheduleBuilder = CronScheduleBuilder.cronSchedule
   (scheduleJob.getCronExpression())
7.     .withMisfireHandlingInstructionDoNothing();
8.
9. //根据cron, 构建一个CronTrigger
10. CronTrigger trigger =
   TriggerBuilder.newTrigger().withIdentity(getTriggerKey(scheduleJob.getJ
   obId())).
11.     withSchedule(scheduleBuilder).build();
12.
13. //放入参数, 运行时的方法可以获取
14. jobDetail.getJobDataMap().put(ScheduleJobEntity.JOB_PARAM_KEY, new
   Gson().toJson(scheduleJob));
15.
16. //把任务添加到Quartz中
17. scheduler.scheduleJob(jobDetail, trigger);

```

把任务添加到 Quartz 后, 等cron定义的时间周期到了, 就会执行 ScheduleJob 类的 executeInternal 方法, ScheduleJob 代码如下所示:

```

1. public class ScheduleJob extends QuartzJobBean {
2.     private Logger logger = LoggerFactory.getLogger(getClass());
3.     private ExecutorService service =
   Executors.newSingleThreadExecutor();
4.
5.     @Override
6.     protected void executeInternal(JobExecutionContext context) throws
   JobExecutionException {
7.         //获取job里的参数, 创建job时, 传进去的ScheduleJobEntity对象
8.         String jsonJob = context.getMergedJobDataMap().getString(Schedu
   leJobEntity.JOB_PARAM_KEY);
9.         ScheduleJobEntity scheduleJob = new Gson().fromJson(jsonJob, Sc
   heduleJobEntity.class);
10.
11.         //获取scheduleJobLogService

```

```

12.         ScheduleJobLogService scheduleJobLogService = (ScheduleJobLogSe
rvice) SpringContextUtils.getBean("scheduleJobLogService");
13.
14.         //数据库保存执行记录
15.         ScheduleJobLogEntity log = new ScheduleJobLogEntity();
16.         log.setJobId(scheduleJob.getJobId());
17.         log.setBeanName(scheduleJob.getBeanName());
18.         log.setMethodName(scheduleJob.getMethodName());
19.         log.setParams(scheduleJob.getParams());
20.         log.setCreateTime(new Date());
21.
22.         //任务开始时间
23.         long startTime = System.currentTimeMillis();
24.
25.         try {
26.             //执行任务，这步是关键
27.             logger.info("任务准备执行，任务ID：" + scheduleJob.getJobId())
;
28.             ScheduleRunnable task = new ScheduleRunnable(scheduleJob.ge
tBeanName(),
29.                 scheduleJob.getMethodName(), scheduleJob.getParams(
));
30.             Future<?> future = service.submit(task);
31.
32.             future.get();
33.
34.             //任务执行总时长
35.             long times = System.currentTimeMillis() - startTime;
36.             log.setTimes((int)times);
37.             //任务状态    0：成功    1：失败
38.             log.setStatus(0);
39.
40.             logger.info("任务执行完毕，任务ID：" + scheduleJob.getJobId()
+ "    总共耗时：" + times + "毫秒");
41.         } catch (Exception e) {
42.             logger.error("任务执行失败，任务ID：" + scheduleJob.getJobId()
, e);
43.
44.             //任务执行总时长
45.             long times = System.currentTimeMillis() - startTime;
46.             log.setTimes((int)times);
47.
48.             //任务状态    0：成功    1：失败
49.             log.setStatus(1);
50.             log.setError(StringUtils.substring(e.toString(), 0, 2000));

```

```

51.         }finally {
52.             scheduleJobLogService.save(log);
53.         }
54.     }
55. }

```

我们搞了一个线程，用来执行定时任务。具体执行是在ScheduleRunnable类里，通过Java反射，执行对应方法的，如下所示：

```

1.  public class ScheduleRunnable implements Runnable {
2.      private Object target;
3.      private Method method;
4.      private String params;
5.
6.      public ScheduleRunnable(String beanName, String methodName, String
params) throws NoSuchMethodException, SecurityException {
7.          //获取spring bean
8.          this.target = SpringContextUtils.getBean(beanName);
9.          this.params = params;
10.
11.          if(StringUtils.isNotBlank(params)){
12.              this.method = target.getClass().getDeclaredMethod(methodName, String.class);
13.          }else{
14.              this.method = target.getClass().getDeclaredMethod(methodName);
15.          }
16.      }
17.
18.      @Override
19.      public void run() {
20.          try {
21.              ReflectionUtils.makeAccessible(method);
22.              if(StringUtils.isNotBlank(params)){
23.                  method.invoke(target, params);
24.              }else{
25.                  method.invoke(target);
26.              }
27.          }catch (Exception e) {
28.              throw new RuntimeException("执行定时任务失败", e);
29.          }
30.      }
31.  }

```

4.4. 云存储模块

图片、文件上传，使用的是七牛、阿里云、腾讯云的存储服务，不能上传到本地服务器。上传到本地服务器，不利于维护，访问速度慢等缺点，所以推荐使用云存储服务。

4.4.1. 七牛的配置

如果没有七牛账号，则需要注册七牛账号，才能进行配置，下面演示注册七牛账号并配置，步骤如下：

1. 注册七牛账号，并登录后，再创建七牛空间，如下图：

产品列表

数据统计 文档中心 工单 站内信 个人面板

新建存储空间

搜索存储空间

存储空间列表

华东

renren

跨区域同步管理

对象存储

融合 CDN

SSL 证书服务

数据处理

直播云服务

存储空间名称

存储空间名称作为唯一的 Bucket 标识符，遇到冲突请更换名称。名称由 4 ~ 63 个字符组成，可包含字母、数字、中划线。

ios-app

存储区域

北美区域尚未支持自定义数据处理服务，一旦创建区域无法修改，请谨慎选择。

华东 华北 华南 北美

访问控制

公开和私有仅对 Bucket 的读文件生效，修改、删除、写入等对 Bucket 的操作均需要拥有者的授权才能进行操作。

公开空间 私有空间

取消 确定创建

2. 进入管理后端，填写七牛配置信息，如下图：

人人快速开发平台

≡ 欢迎 admin

导航菜单

系统管理

管理员管理

部门管理

角色管理

菜单管理

SQL监控

定时任务

参数管理

文件上传

系统日志

首页 > 文件上传

云存储配置

存储类型

七牛

阿里云

腾讯云

免费申请(七牛)10GB储存空间

域名

http://7xqbwh.dl1.z0.glb.clouddn.com

路径前缀

upload

AccessKey

NrgMfABZxWLo5B-YYsJoE8-AZ1EISdi1Z3ubLOeZ

SecretKey

ulwJHevMRWU0VLxFvgy0tAcOdGqasdtVIJkdy6vV

空间名

ios-app

确定

返回

必填项有域名、AccessKey、SecretKey、空间名。其中，空间名就是才创建的空间名 `ios-app`，填进去就可以了。域名、AccessKey、SecretKey可以通过下图找到：

资源主页

个人中心

财务统计

对象存储

融合 CDN

SSL 证书服务

数据处理

直播云服务

产品列表

新建存储空间

搜索存储空间

存储空间列表

华东

ios-app

renren

跨区域同步管理

数据统计

文档中心

工单

站内信

个人面板

los-app

空间概览

数据统计

内容管理

镜像存储

绑定域名

样式分隔符设置

图片样式

空间授权

生命周期

空间设置

标准存储

低频存储

文件存储

0 B

9月01日 9月03日 9月05日 9月07日

API 请求

GET

PUT

测试域名

△ 此类测试域名，限总流量，限单 IP 访问频率，限速，仅供测试使用，不能用于自定义域名的 CNAME 相关文档

ovybgq0n2.bkt.clouddn.com

域名

融合 CDN 加速域名

自定义域名

立即绑定一个域名

资源主页

个人中心

财务统计

对象存储

融合 CDN

SSL 证书服务

智能多媒体

直播云服务

产品列表

数据统计 | 文档中心 | 工单 | 站内信 | 个人面板

个人中心 > 密钥管理

个人信息 密钥管理 安全设置 提醒设置 操作日志 关联账户 邀请好友

一个账号最多拥有两对密钥(Access/Secret Key)；更换密钥时，请创建第二个密钥；删除密钥前须停用；出于安全考虑，建议您周期性地更换密钥。您可以查看更多 [安全使用密钥建议](#)。

创建时间	AccessKey/SecretKey	状态	操作
2015-07-06	AK: NrgMfABZxWLo5B-YY5joE8-AZ1EISdi1Z3ubLOeZ	使用中	停用
	SK: ulwJHevMRWU0VLxFvgY0tAcOdGqasdtVUkdy6vV		

4.4.2. 阿里云的配置

- 进入管理后端，填写阿里云配置信息，如下图：

导航菜单

系统管理

管理员管理

部门管理

角色管理

菜单管理

SQL监控

定时任务

参数管理

文件上传

系统日志

首页 > 文件上传

云存储配置

存储类型

☐ 七牛

☒ 阿里云

☐ 腾讯云

域名

阿里云绑定的域名

路径前缀

不设置默认为空

EndPoint

阿里云EndPoint

AccessKeyId

阿里云AccessKeyId

AccessKeySecret

阿里云AccessKeySecret

BucketName

阿里云BucketName

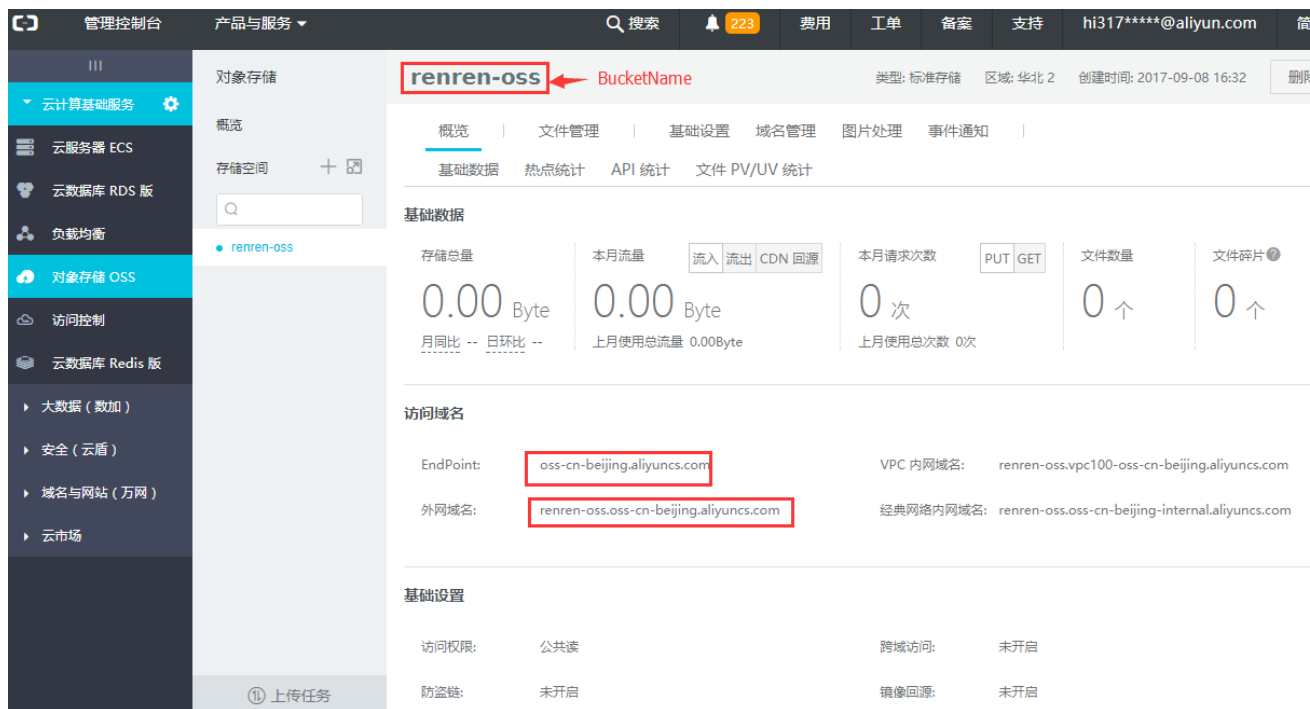
确定

返回

- 进去阿里云管理后台，并创建Bucket，如下图：



- 通过下面的界面，可以找到域名、BucketName、EndPoint



- 通过下面的界面，可以找到AccessKeyId、AccessKeySecret



4.4.3. 腾讯云的配置

- 进入管理后端，填写腾讯云配置信息，如下图：

人人快速开发平台

欢迎 admin

首页 > 文件上传

云存储配置

存储类型 ☐ 七牛 ☐ 阿里云 ☒ 腾讯云

域名

路径前缀

AppId

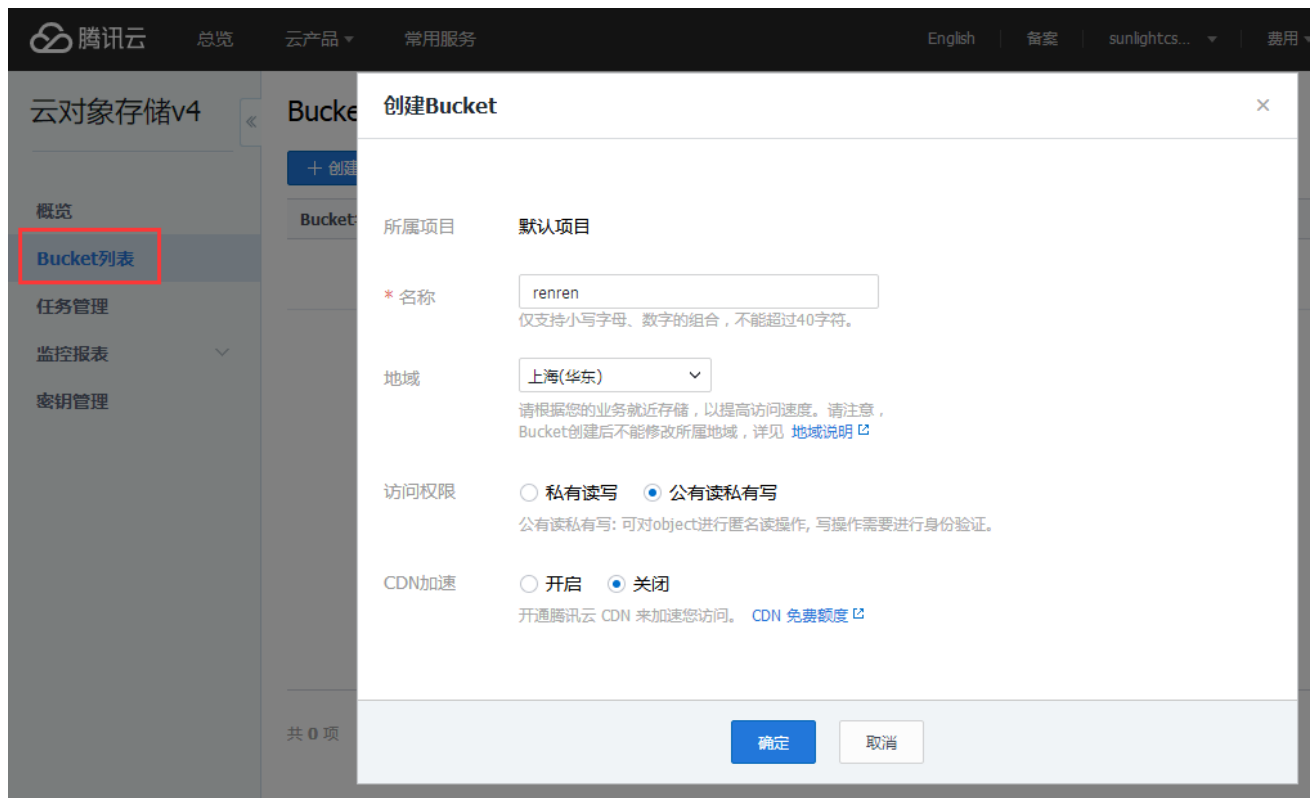
SecretId

SecretKey

BucketName

Bucket所属地区

- 进去腾讯云管理后台，并创建Bucket，如下图：



- 通过下面的界面，可以找到域名、BucketName、Bucket所属地区





- 通过下面的界面，可以找到AppId、SecretId、SecretKey



4.4.4. 源码分析

- 本项目的文件上传，使用的是七牛、阿里云、腾讯云，则需要引入他们的SDK，如下：

```
1. <dependency>
2.   <groupId>com.qiniu</groupId>
3.   <artifactId>qiniu-java-sdk</artifactId>
4.   <version>${qiniu.version}</version>
5. </dependency>
```

```

6.     <dependency>
7.         <groupId>com.aliyun.oss</groupId>
8.         <artifactId>aliyun-sdk-oss</artifactId>
9.         <version>${aliyun.oss.version}</version>
10.    </dependency>
11.    <dependency>
12.        <groupId>com.qcloud</groupId>
13.        <artifactId>cos_api</artifactId>
14.        <version>${qcloud.cos.version}</version>
15.        <exclusions>
16.            <exclusion>
17.                <groupId>org.slf4j</groupId>
18.                <artifactId>slf4j-log4j12</artifactId>
19.            </exclusion>
20.        </exclusions>
21.    </dependency>

```

- 定义抽象类 `CloudStorageService`，用来声明上传的公共接口，如下所示：

```

1.    public abstract class CloudStorageService {
2.        /** 云存储配置信息 */
3.        CloudStorageConfig config;
4.
5.        /**
6.         * 文件路径
7.         * @param prefix 前缀
8.         * @return 返回上传路径
9.         */
10.       public String getPath(String prefix) {
11.           //生成uuid
12.           String uuid = UUID.randomUUID().toString().replaceAll("-", "");
13.           //文件路径
14.           String path = DateUtils.format(new Date(), "yyyyMMdd") + "/" +
uuid;
15.
16.           if (StringUtils.isNotBlank(prefix)) {
17.               path = prefix + "/" + path;
18.           }
19.
20.           return path;
21.       }
22.
23.       /**
24.        * 文件上传

```

```

25.      * @param data      文件字节数组
26.      * @param path      文件路径, 包含文件名
27.      * @return          返回http地址
28.      */
29.      public abstract String upload(byte[] data, String path);
30.
31.      /**
32.      * 文件上传
33.      * @param data      文件字节数组
34.      * @return          返回http地址
35.      */
36.      public abstract String upload(byte[] data);
37.
38.      /**
39.      * 文件上传
40.      * @param inputStream 字节流
41.      * @param path        文件路径, 包含文件名
42.      * @return            返回http地址
43.      */
44.      public abstract String upload(InputStream inputStream, String path)
45.      ;
46.
47.      /**
48.      * 文件上传
49.      * @param inputStream 字节流
50.      * @return            返回http地址
51.      */
51.      public abstract String upload(InputStream inputStream);
52.
53.  }

```

- 七牛上传的实现, 只需继承 `CloudStorageService`, 并实现相应的上传接口, 如下所示:

```

1.  import com.qiniu.common.Zone;
2.  import com.qiniu.http.Response;
3.  import com.qiniu.storage.Configuration;
4.  import com.qiniu.storage.UploadManager;
5.  import com.qiniu.util.Auth;
6.  import io.renren.common.exception.RRException;
7.  import org.apache.commons.io.IOUtils;
8.
9.  public class QiniuCloudStorageService extends CloudStorageService{
10.      private UploadManager uploadManager;
11.      private String token;

```

```
12.
13.     public QiniuCloudStorageService(CloudStorageConfig config){
14.         this.config = config;
15.
16.         //初始化
17.         init();
18.     }
19.
20.     private void init(){
21.         uploadManager = new UploadManager(new Configuration(Zone.autoZone()));
22.         token = Auth.create(config.getQiniuAccessKey(), config.getQiniuSecretKey());
23.         uploadToken(config.getQiniuBucketName());
24.     }
25.
26.     @Override
27.     public String upload(byte[] data, String path) {
28.         try {
29.             Response res = uploadManager.put(data, path, token);
30.             if (!res.isOK()) {
31.                 throw new RuntimeException("上传七牛出错：" + res.toString());
32.             }
33.         } catch (Exception e) {
34.             throw new RuntimeException("上传文件失败，请核对七牛配置信息", e);
35.         }
36.
37.         return config.getQiniuDomain() + "/" + path;
38.     }
39.
40.     @Override
41.     public String upload(InputStream inputStream, String path) {
42.         try {
43.             byte[] data = IOUtils.toByteArray(inputStream);
44.             return this.upload(data, path);
45.         } catch (IOException e) {
46.             throw new RuntimeException("上传文件失败", e);
47.         }
48.     }
49.
50.     @Override
51.     public String upload(byte[] data) {
52.         return upload(data, getPath(config.getQiniuPrefix()));
53.     }
```

```

54.
55.     @Override
56.     public String upload(InputStream inputStream) {
57.         return upload(inputStream, getPath(config.getQiniuPrefix()));
58.     }
59. }

```

- 阿里云上传的实现，只需继承 `CloudStorageService`，并实现相应的上传接口，如下所示：

```

1.  import com.aliyun.oss.OSSClient;
2.  import java.io.ByteArrayInputStream;
3.  import java.io.InputStream;
4.
5.  public class AliyunCloudStorageService extends CloudStorageService{
6.      private OSSClient client;
7.
8.      public AliyunCloudStorageService(CloudStorageConfig config){
9.          this.config = config;
10.
11.          //初始化
12.          init();
13.      }
14.
15.      private void init(){
16.          client = new OSSClient(config.getAliyunEndPoint(), config.getAl
17.          iyunAccessKeyId(),
18.              config.getAliyunAccessKeySecret());
19.      }
20.
21.      @Override
22.      public String upload(byte[] data, String path) {
23.          return upload(new ByteArrayInputStream(data), path);
24.      }
25.
26.      @Override
27.      public String upload(InputStream inputStream, String path) {
28.          try {
29.              client.putObject(config.getAliyunBucketName(), path, inputS
30.              tream);
31.          } catch (Exception e){
32.              throw new RuntimeException("上传文件失败，请检查配置信息", e);
33.          }
34.
35.          return config.getAliyunDomain() + "/" + path;
36.      }
37.  }

```

```

34.     }
35.
36.     @Override
37.     public String upload(byte[] data) {
38.         return upload(data, getPath(config.getAliyunPrefix()));
39.     }
40.
41.     @Override
42.     public String upload(InputStream inputStream) {
43.         return upload(inputStream, getPath(config.getAliyunPrefix()));
44.     }
45. }

```

- 腾讯云上传的实现，只需继承 `CloudStorageService`，并实现相应的上传接口，如下所示：

```

1.  import com.qcloud.cos.COSClient;
2.  import com.qcloud.cos.ClientConfig;
3.  import com.qcloud.cos.request.UploadFileRequest;
4.  import com.qcloud.cos.sign.Credentials;
5.  import net.sf.json.JSONObject;
6.  import org.apache.commons.io.IOUtils;
7.
8.  public class QcloudCloudStorageService extends CloudStorageService{
9.      private COSClient client;
10.
11.      public QcloudCloudStorageService(CloudStorageConfig config){
12.          this.config = config;
13.
14.          //初始化
15.          init();
16.      }
17.
18.      private void init(){
19.          Credentials credentials = new Credentials(config.getQcloudAppId
20.          (), config.getQcloudSecretId(),
21.              config.getQcloudSecretKey());
22.
23.          //初始化客户端配置
24.          ClientConfig clientConfig = new ClientConfig();
25.          //设置bucket所在的区域，华南：gz 华北：tj 华东：sh
26.          clientConfig.setRegion(config.getQcloudRegion());
27.
28.          client = new COSClient(clientConfig, credentials);
29.      }

```

```

29.
30.     @Override
31.     public String upload(byte[] data, String path) {
32.         //腾讯云必需要以"/"开头
33.         if(!path.startsWith("/")) {
34.             path = "/" + path;
35.         }
36.
37.         //上传到腾讯云
38.         UploadFileRequest request = new UploadFileRequest(config.getQcloudBucketName(), path, data);
39.         String response = client.uploadFile(request);
40.
41.         JSONObject jsonObject = JSONObject.fromObject(response);
42.         if(jsonObject.getInt("code") != 0) {
43.             throw new RuntimeException("文件上传失败, " + jsonObject.getString("message"));
44.         }
45.
46.         return config.getQcloudDomain() + path;
47.     }
48.
49.     @Override
50.     public String upload(InputStream inputStream, String path) {
51.         try {
52.             byte[] data = IOUtils.toByteArray(inputStream);
53.             return this.upload(data, path);
54.         } catch (IOException e) {
55.             throw new RuntimeException("上传文件失败", e);
56.         }
57.     }
58.
59.     @Override
60.     public String upload(byte[] data) {
61.         return upload(data, getPath(config.getQcloudPrefix()));
62.     }
63.
64.     @Override
65.     public String upload(InputStream inputStream) {
66.         return upload(inputStream, getPath(config.getQcloudPrefix()));
67.     }
68. }

```

- 对外提供了OSSFactory工厂，可方便业务的调用，如下所示：


```

1.  public final class OSSFactory {
2.      private static SysConfigService sysConfigService;
3.
4.      static {
5.          OSSFactory.sysConfigService = (SysConfigService)
SpringContextUtils.getBean("sysConfigService");
6.      }
7.
8.      public static CloudStorageService build(){
9.          //获取云存储配置信息
10.         CloudStorageConfig config = sysConfigService.getConfigObject(Co
nfigConstant.CLOUD_STORAGE_CONFIG_KEY, CloudStorageConfig.class);
11.
12.         if(config.getType() == Constant.CloudService.QINIU.getValue()){
13.             return new QiniuCloudStorageService(config);
14.         }else if(config.getType() == Constant.CloudService.ALIYUN.getVa
lue()){
15.             return new AliyunCloudStorageService(config);
16.         }else if(config.getType() == Constant.CloudService.QCLOUD.getVa
lue()){
17.             return new QcloudCloudStorageService(config);
18.         }
19.
20.         return null;
21.     }
22.
23. }

```

- 文件上传的例子，如下：

```

1.  @RequestMapping("/upload")
2.  public R upload(@RequestParam("file") MultipartFile file) throws Except
ion {
3.      if (file.isEmpty()) {
4.          throw new RRException("上传文件不能为空");
5.      }
6.
7.      //上传文件，并返回文件的http地址
8.      String url = OSSFactory.build().upload(file.getBytes());
9.  }

```

4.5. APP模块

APP模块，是针对APP使用的，如IOS、Android等，主要是解决用户认证的问题。

4.5.1. APP的使用

APP的设计思路：用户通过APP，输入手机号、密码登录后，系统会生成与登录用户一一对应的token，用户调用需要登录的接口时，只需把token传过来，服务端就知道是谁在访问接口，token如果过期，则拒绝访问，从而保证系统的安全性。

使用很简单，看看下面的例子，就会使用了。仔细观察，我们会发现，有2个自定义的注解。其中，@LoginUser注解是获取当前登录用户的信息，有哪些信息，下面会分析的。@Login注解则是需要用户认证，没有登录的用户，不能访问该接口。

```
1.  import io.renren.modules.app.annotation.Login;
2.  import io.renren.modules.app.annotation.LoginUser;
3.
4.  @RestController
5.  @RequestMapping("/app")
6.  public class ApiTestController {
7.
8.      /**
9.       * 获取用户信息
10.      */
11.     @Login
12.     @GetMapping("userInfo")
13.     public R userInfo(@LoginUser UserEntity user) {
14.         return R.ok().put("user", user);
15.     }
16.
17.     /**
18.      * 获取用户ID
19.      */
20.     @Login
21.     @GetMapping("userId")
22.     public R userInfo(@RequestAttribute("userId") Integer userId) {
23.         return R.ok().put("userId", userId);
24.     }
25.
26.     /**
27.      * 忽略Token验证测试
```

```
28.         */
29.         @GetMapping("notToken")
30.         public R notToken() {
31.             return R.ok().put("msg", "无需token也能访问。。。");
32.         }
33.
34.     }
```

4.5.2. 源码分析

- 我们先来看看，APP用户登录的时候，都干了那些事情，如下所示：

```
1.     @RestController
2.     @RequestMapping("/app")
3.     public class ApiLoginController {
4.         @Autowired
5.         private UserService userService;
6.         @Autowired
7.         private JwtUtils jwtUtils;
8.
9.         /**
10.          * 登录
11.          */
12.         @PostMapping("login")
13.         public R login(String mobile, String password) {
14.             Assert.isBlank(mobile, "手机号不能为空");
15.             Assert.isBlank(password, "密码不能为空");
16.
17.             //用户登录
18.             long userId = userService.login(mobile, password);
19.
20.             //生成token
21.             String token = jwtUtils.generateToken(userId);
22.
23.             Map<String, Object> map = new HashMap<>();
24.             map.put("token", token);
25.             map.put("expire", jwtUtils.getExpire());
26.
27.             return R.ok(map);
28.         }
29.
30.     }
31.
```

```
32. -----
33.
34.
35.
36. /**
37.  * jwt工具类
38.  */
39. @ConfigurationProperties(prefix = "renren.jwt")
40. @Component
41. public class JwtUtils {
42.     private Logger logger = LoggerFactory.getLogger(getClass());
43.
44.     private String secret;
45.     private long expire;
46.     private String header;
47.
48.     /**
49.      * 生成jwt token
50.      */
51.     public String generateToken(long userId) {
52.         Date nowDate = new Date();
53.         //过期时间
54.         Date expireDate = new Date(nowDate.getTime() + expire * 1000);
55.
56.         return Jwts.builder()
57.             .setHeaderParam("typ", "JWT")
58.             .setSubject(userId+"")
59.             .setIssuedAt(nowDate)
60.             .setExpiration(expireDate)
61.             .signWith(SignatureAlgorithm.HS512, secret)
62.             .compact();
63.     }
64.
65.     public Claims getClaimByToken(String token) {
66.         try {
67.             return Jwts.parser()
68.                 .setSigningKey(secret)
69.                 .parseClaimsJws(token)
70.                 .getBody();
71.         } catch (Exception e) {
72.             logger.debug("validate is token error ", e);
73.             return null;
74.         }
75.     }
76. }
```

```

77.      /**
78.       * token是否过期
79.       * @return true: 过期
80.       */
81.      public boolean isTokenExpired(Date expiration) {
82.          return expiration.before(new Date());
83.      }
84.
85.      public String getSecret() {
86.          return secret;
87.      }
88.
89.      public void setSecret(String secret) {
90.          this.secret = secret;
91.      }
92.
93.      public long getExpire() {
94.          return expire;
95.      }
96.
97.      public void setExpire(long expire) {
98.          this.expire = expire;
99.      }
100.
101.      public String getHeader() {
102.          return header;
103.      }
104.
105.      public void setHeader(String header) {
106.          this.header = header;
107.      }
108.  }

```

我们从上面的代码，可以看到，用户每次登录的时候，都会生成一个唯一的token，这个token是通过jwt生成的。

- APP模块的核心配置，如下所示：

```

1.      import io.renren.modules.api.interceptor.AuthorizationInterceptor;
2.      import
3.      io.renren.modules.api.resolver.LoginUserHandlerMethodArgumentResolver;
4.      import org.springframework.beans.factory.annotation.Autowired;
5.      import org.springframework.context.annotation.Configuration;
6.      import

```

```

6.  org.springframework.web.method.support.HandlerMethodArgumentResolver;
    import
7.  org.springframework.web.servlet.config.annotation.InterceptorRegistry;
    import
    org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
8.
9.  @Configuration
10. public class WebMvcConfig extends WebMvcConfigurerAdapter {
11.     @Autowired
12.     private AuthorizationInterceptor authorizationInterceptor;
13.     @Autowired
14.     private LoginUserHandlerMethodArgumentResolver
loginUserHandlerMethodArgumentResolver;
15.
16.     @Override
17.     public void addInterceptors(InterceptorRegistry registry) {
18.         registry.addInterceptor(authorizationInterceptor).addPathPatterns("/app/**");
19.     }
20.
21.     @Override
22.     public void
addArgumentResolvers(List<HandlerMethodArgumentResolver>
argumentResolvers) {
23.         argumentResolvers.add(loginUserHandlerMethodArgumentResolver);
24.     }
25. }

```

我们可以看到，配置了个Interceptor，用来拦截 /app 开头的请求，拦截后，会到 AuthorizationInterceptor类preHandle方法处理。只有以 /app 开头的请求，API模块认证才会起作用，如果要以 /api 开头，则需要修改此处。还配置了argumentResolver，别忽略了啊，下面会讲解。

温馨提示，别忘了配置shiro，不然会被shiro拦截掉的，如下所示：

```

1.  @Configuration
2.  public class ShiroConfig {
3.      @Bean("shiroFilter")
4.      public ShiroFilterFactoryBean shirFilter(SecurityManager
securityManager) {
5.
6.          //部分代码省略...
7.

```

```

8.         Map<String, String> filterMap = new LinkedHashMap<>();
9.         //让shiro放过, 以/app开头的请求
10.        filterMap.put("/app/**", "anon");
11.
12.        //部分代码省略...
13.
14.        shiroFilter.setFilterChainDefinitionMap(filterMap);
15.
16.        return shiroFilter;
17.    }
18. }

```

- 分析AuthorizationInterceptor类, 我们可以发现, 拦截 /app 开头的请求后, 都干了些什么, 如下所示:

```

1.
2.     import io.jsonwebtoken.Claims;
3.     import io.renren.common.exception.RRException;
4.     import io.renren.modules.app.utils.JwtUtils;
5.     import io.renren.modules.app.annotation.Login;
6.     import org.apache.commons.lang.StringUtils;
7.     import org.springframework.beans.factory.annotation.Autowired;
8.     import org.springframework.http.HttpStatus;
9.     import org.springframework.stereotype.Component;
10.    import org.springframework.web.method.HandlerMethod;
11.    import
    org.springframework.web.servlet.handler.HandlerInterceptorAdapter;
12.
13.    import javax.servlet.http.HttpServletRequest;
14.    import javax.servlet.http.HttpServletResponse;
15.
16.    /**
17.     * 权限(Token)验证
18.     */
19.    @Component
20.    public class AuthorizationInterceptor extends
    HandlerInterceptorAdapter {
21.        @Autowired
22.        private JwtUtils jwtUtils;
23.
24.        public static final String USER_KEY = "userId";
25.
26.        @Override
27.        public boolean preHandle(HttpServletRequest request, HttpServletResponse

```

```

28.    ponse response, Object handler) throws Exception {
29.         Login annotation;
30.         if(handler instanceof HandlerMethod) {
31.             annotation = ((HandlerMethod) handler).getMethodAnnotation(
Login.class);
32.         }else{
33.             return true;
34.         }
35.         if(annotation == null){
36.             return true;
37.         }
38.
39.         //获取用户凭证
40.         String token = request.getHeader(jwtUtils.getHeader());
41.         if(StringUtils.isBlank(token)){
42.             token = request.getParameter(jwtUtils.getHeader());
43.         }
44.
45.         //凭证为空
46.         if(StringUtils.isBlank(token)){
47.             throw new RuntimeException(jwtUtils.getHeader() + "不能为空",
HttpStatus.UNAUTHORIZED.value());
48.         }
49.
50.         Claims claims = jwtUtils.getClaimByToken(token);
51.         if(claims == null || jwtUtils.isTokenExpired(claims.getExpirati
on())){
52.             throw new RuntimeException(jwtUtils.getHeader() + "失效, 请重新登录
", HttpStatus.UNAUTHORIZED.value());
53.         }
54.
55.         //设置userId到request里, 后续根据userId, 获取用户信息
56.         request.setAttribute(USER_KEY, Long.parseLong(claims.getSubject
()));
57.
58.         return true;
59.     }
60. }

```

我们可以发现, 进入 `/app` 请求的接口之前, 会判断请求的接口, 是否加了@Login注解(需要token认证), 如果没有@Login注解, 则不验证token, 可以直接访问接口。如果有@Login注解, 则需要验证token的正确性, 并把userId放到request的USER_KEY里, 后续会用到。

- 此时，@Login注解的作用，相信大家都明白了。再看看下面的代码，加了@loginUser注解后，user对象里，就变成当前登录用户的信息，这是什么时候设置进去的呢？

```
1.  /**
2.   * 获取用户信息
3.   */
4.  @GetMapping("userInfo")
5.  public R userInfo(@LoginUser UserEntity user) {
6.      return R.ok().put("user", user);
7.  }
```

- 设置user对象进去，其实是在LoginUserHandlerMethodArgumentResolver里干的，LoginUserHandlerMethodArgumentResolver是我们自定义的参数转换器，只要实现HandlerMethodArgumentResolver接口即可，代码如下所示：

```
1.  import io.renren.modules.api.annotation.LoginUser;
2.  import io.renren.modules.api.entity.UserEntity;
3.  import io.renren.modules.api.interceptor.AuthorizationInterceptor;
4.  import io.renren.modules.api.service.UserService;
5.  import org.springframework.beans.factory.annotation.Autowired;
6.  import org.springframework.core.MethodParameter;
7.  import org.springframework.stereotype.Component;
8.  import org.springframework.web.bind.support.WebDataBinderFactory;
9.  import org.springframework.web.context.request.NativeWebRequest;
10. import org.springframework.web.context.request.RequestAttributes;
11. import
12.     org.springframework.web.method.support.HandlerMethodArgumentResolver;
13. import org.springframework.web.method.support.ModelAndViewContainer;
14.
15. @Component
16. public class LoginUserHandlerMethodArgumentResolver implements
17.     HandlerMethodArgumentResolver {
18.
19.     @Autowired
20.     private UserService userService;
21.
22.     @Override
23.     public boolean supportsParameter(MethodParameter parameter) {
24.         //如果方法的参数是UserEntity，且参数前面有@loginUser注解，则进入
25.         resolveArgument方法，进行处理
26.         return parameter.getParameterType().isAssignableFrom(UserEntity
27.             .class) && parameter.hasParameterAnnotation(LoginUser.class);
28.     }
29. }
```

```

25.         @Override
26.         public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer container,
27.                                     NativeWebRequest request, WebDataBinderFactory factory) throws Exception {
28.             //获取用户ID, 之前设置进去的, 还有印象吧
29.             Object object = request.getAttribute(AuthorizationInterceptor.USER_KEY, RequestAttributes.SCOPE_REQUEST);
30.             if(object == null){
31.                 return null;
32.             }
33.
34.             //通过userId, 获取用户信息
35.             UserEntity user = userService.queryObject((Long)object);
36.
37.             //把当前用户信息, 设置到UserEntity参数的user对象里
38.             return user;
39.         }
40.     }

```

5. 生产环境部署

5.1. 常规部署

Spring Boot项目，推荐打成jar包的方式，部署到服务器上。

- Spring Boot内置了Tomcat，可配置Tomcat的端口号、初始化线程数、最大线程数、连接超时时长、https等等，如下所示：

```

1.     server:
2.         tomcat:
3.             uri-encoding: UTF-8
4.             max-threads: 1000
5.             min-spare-threads: 20
6.             connection-timeout: 5000
7.             port: 80
8.             context-path: /renren-fast
9.             ssl:
10.                 key-store: classpath:.keystore

```

```
11.     key-store-type: JKS
12.     key-password: 123456
13.     key-alias: tomcat
```

- 当然，还可以指定jvm的内存大小，如下所示：

```
1.     java -Xms4g -Xmx4g -Xmn1g -server -jar renren-fast-1.2.0.jar
```

- 在windows下部署，只需打开cmd窗口，输入如下命令：

```
1.     java -jar renren-fast-1.2.0.jar --spring.profiles.active=pro
```

- 在Linux下部署，只需输入如下命令，即可在Linux后台运行，还可以放到/etc/rc.local里，每次重启Linux时，项目都会自动起来：

```
1.     nohup java -jar renren-fast-1.2.0.jar --spring.profiles.active=pro > re
      nren.log &
```

5.2. war包部署

war包的部署，也很方便，只是不推荐这种方式。

- 在项目的根目录下，执行【mvn clean package -f pom-war.xml】命令，打成war包
- 把生成的war包，放在tomcat【8.0+】的webapps目录下面，再启动tomcat即可

5.3. docker部署

- 安装docker环境

```
1.     #安装docker
2.     [root@mark ~]# curl -sSL https://get.docker.com/ | sh
3.
4.     #启动docker
```

```
5. [root@mark ~]# service docker start
6.
7. #查看docker版本信息
8. [root@mark ~]# docker version
9. Client:
10.  Version:      17.07.0-ce
11.  API version:  1.31
12.  Go version:   go1.8.3
13.  Git commit:   8784753
14.  Built:        Tue Aug 29 17:42:01 2017
15.  OS/Arch:      linux/amd64
16.
17. Server:
18.  Version:      17.07.0-ce
19.  API version:  1.31 (minimum version 1.12)
20.  Go version:   go1.8.3
21.  Git commit:   8784753
22.  Built:        Tue Aug 29 17:43:23 2017
23.  OS/Arch:      linux/amd64
24.  Experimental: false
```

- 还需要准备java、maven环境，请自行安装
- 通过maven插件，构建docker镜像

```
1. #打包并构建项目镜像
2. [root@mark renren-fast]# mvn clean package docker:build
3. #省略打包log...
4. [INFO] Building image renren/fast
5. Step 1/6 : FROM java:8
6. ----> d23bdf5b1b1b
7. Step 2/6 : EXPOSE 8080
8. ----> Using cache
9. ----> 8e33aadb2c18
10. Step 3/6 : VOLUME /tmp
11. ----> Using cache
12. ----> c5dc0c509062
13. Step 4/6 : ADD renren-fast-1.2.0.jar /app.jar
14. ----> 831bc3ca84bc
15. Step 5/6 : RUN bash -c 'touch /app.jar'
16. ----> Running in fe3ef9343e4c
17. ----> b3d6dd6fc297
18. Removing intermediate container fe3ef9343e4c
19. Step 6/6 : ENTRYPOINT java -jar /app.jar
20. ----> Running in 89adce4ae167
```

```

21.    ---> a4ae60970a77
22.    Removing intermediate container 89adce4ae167
23.    ProgressMessage{id=null, status=null, stream=null, error=null,
24.    progress=null, progressDetail=null}
25.    Successfully built a4ae60970a77
26.    Successfully tagged renren/fast:latest
27.    #查看镜像
28.    [root@mark renren-fast]# docker images
29.    REPOSITORY          TAG                 IMAGE ID           CREATED
30.    SIZE
31.    renren/fast         latest            a4ae60970a77      14 seconds
32.    ago                714MB
33.    java                8                 d23bdf5b1b1b      7 months ago
34.    643MB

```

- 安装docker-compose，用来管理容器

```

1.    #下载地址：https://github.com/docker/compose/releases
2.
3.    #下载docker-compose
4.    [root@mark renren-fast]# curl -L
5.    https://github.com/docker/compose/releases/download/1.16.1/docker-comp
6.    ose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
7.
8.    #增加可执行权限
9.    [root@mark renren-fast]# chmod +x /usr/local/bin/docker-compose
10.
11.    #查看版本信息
12.    [root@mark renren-fast]# docker-compose version
13.    docker-compose version 1.16.1, build 6dlac21
14.    docker-py version: 2.5.1
15.    CPython version: 2.7.13
16.    OpenSSL version: OpenSSL 1.0.1t  3 May 2016

```

如果下载不了，可以用迅雷

将https://github.com/docker/compose/releases/download/1.16.1/docker-compose-Linux-x86_64下载到本地，再上传到服务器

- 通过docker-compose，启动项目，如下所示：

```

1.    #启动项目
2.    [root@mark renren-fast]# docker-compose up -d

```

```
3. Creating network "renrenfast_default" with the default driver
4. Creating renrenfast_campus_1 ...
5. Creating renrenfast_campus_1 ... done
6.
7. #查看启动的容器
8. [root@mark renren-fast]# docker ps
9. CONTAINER ID          IMAGE                COMMAND              CREATED
   STATUS                PORTS               NAMES
10. f4e3fcdd8dd4          renren/fast         "java -jar /app.jar" 55 secon
   ds ago               Up 3 seconds        0.0.0.0:8080->8080/tcp
   renrenfast_renren-fast_1
11.
12. #停掉并删除, docker-compose管理的容器
13. [root@mark renren-fast]# docker-compose down
14. Stopping renrenfast_renren-fast_1 ... done
15. Removing renrenfast_renren-fast_1 ... done
16. Removing network renrenfast_default
```
